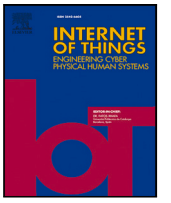
Internet of Things 29 (2025) 101436

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot



Research article

System-wide IoT design and programming: Patterns for decentralised collective processes

Roberto Casadei

Alma Mater Studiorum—Università di Bologna, Department of Computer Science and Engineering, Via dell'Università 50, Cesena, 47521, Italy

ARTICLE INFO

Dataset link: https://zenodo.org/records/11210637

Keywords:
Internet of Things
Collective intelligence
Distributed computing
Decentralised systems
Macro-programming
Design patterns

ABSTRACT

The Internet of Things promotes a view of large-scale deployments of devices able to compute, communicate, and interact with their surrounding environment. In this context, one significant challenge revolves around designing and programming collective processes, i.e., durable activities involving the collaboration of large groups of devices. Examples of collective processes include distributed sensing, collective decision-making, collective movement/transport, and adaptive maintenance of system-level structures. To address the issues involved in developing such kinds of system-wide behaviours, research has proposed multiple approaches, abstractions, and algorithmic solutions. In particular, the approach of *aggregate processes* has emerged as a promising formal technique for programming collective processes by a macro-level perspective while supporting decentralisation, abstraction, and resilience. In order to characterise (i) previous work on aggregate processes, (ii) the usages and applications that this technique may foster, and (iii) draw general design insights in the realm of collective computing, this article provides a characterisation of common problems and solutions based on aggregate processes. What results is a catalogue of design patterns for decentralised collective processes. Specifically, we provide a taxonomy of patterns, describe each pattern in a schematic form, and discuss the implications for the design of collective processes for the Internet of Things and related scenarios.

1. Introduction

The Internet of Things (IoT) [1] provides a setting where addressable and interconnected things, backed by smart devices, can collaborate to support a plethora of situated services and applications. This paradigm fosters a view that goes beyond individual devices, namely one focussing on *systems* [2], *systems of systems* [3], *swarms* [4], or *collectives* [5] of interacting devices.

One kind of activities that may be executed on an IoT system is given by so-called *collective processes* [4,6,7], namely durable activities involving the collaboration of dynamic and possibly large groups of devices. Examples of collective processes may include monitoring and surveillance tasks [8,9], allocation tasks [10], collective movement [11], swarm control [4,12], resilient formation of device clusters [13,14], spatial coordination [15], and human-machine collaborative tasks in smart societies [16,17].

The focus of this manuscript is on the design and implementation of programs driving collective processes by controlling the decentralised activity and interaction of an underlying set of IoT devices. We also refer to this as *system-wide programming*, since the idea is to program the system "as a whole", rather than the individual devices that are then integrated into the system. Indeed, research has been devoted to investigating programming models and languages for swarms [12,18] and collective systems [19], sometimes grouped under the umbrella of *macro-programming* [20,21]. There also exist small catalogues of patterns

E-mail address: roby.casadei@unibo.it.

https://doi.org/10.1016/j.iot.2024.101436
Received 21 May 2024; Received in revised form 18 October 2024; Accepted 13 November 2024
Available online 11 December 2024

for collective behaviour: e.g., in the context of spatial computing [22], biologically-inspired coordination [23,24], and field-based coordination [25]. These cover basic self-organisation patterns like gradients [26], information flows [27], distributed collection [28], and leader election [29]. Patterns are very important in software engineering, for they distill experience and knowledge about reasoned solutions for recurrent problems. Therefore, in this work, we provide a catalogue of patterns for decentralised collective processes. Most specifically, we provide the following contributions:

- we define a taxonomy of design and coding patterns for collective processes, based on the analysis of related taxonomies/catalogues [18,23,26,30] and previous work on aggregate processes (e.g. [6,31]);
- we identify and characterise several patterns fitting the taxonomy, describe them in a structured format (*Name, Aliases, Intent, Context and Examples, Description, Implementation, Discussion*), and accompany them with an implementation schema in the ScaFi aggregate programming language [32];
- we provide a permanent open-source artefact with reproducible simulations exercising the patterns [33] and a case study [15] showing the applications of multiple collective process patterns in a single system.

In particular, a major novel contribution lies in addressing the *programming* viewpoint of collective behaviour, with actual *coding* patterns besides design. Moreover, the code is expressed as valid code in Scala, a mainstream programming language which hosts the ScaFi library of aggregate computing.

The remainder of the manuscript is organised as follows. Section 2 covers context, related work, and motivations. Section 3 provides background, and specifically describes a formal operational model for aggregate processes, together with the basics of programming in the ScaFi language. Section 4 presents the method adopted in this study, detailing how patterns are identified, classified, and described. Section 5 provides the main contribution, namely the catalogue of patterns. Section 6 presents a case study where multiple proposed patterns are applied into a coherent solution. Section 7 discusses the contribution by different perspectives. Section 8 sums up the contribution and offers conclusive thoughts.

2. Related work and motivation

This section discusses related work and provides motivation for this paper contribution. First, it discusses design patterns and pattern catalogues in software engineering (Section 2.1), to provide general background for framing the contribution. Then, it zooms in on design patterns specific for the IoT context (Section 2.2) and for expressing collective behaviour (Section 2.3). Finally, in the light of the reviewed literature, it clarifies the motivation of the work and the positioning of the contribution (Section 2.4).

2.1. Patterns and pattern catalogues

Design patterns distill knowledge by documenting solutions to recurrent problems in specific contexts [34]. Specifically, a pattern has a name and is described in a way that documents important aspects like the intents of the pattern, its applicability, the "forces" that constrain or affect its solution, the abstract structure of the solution, and its consequences [34]. Popular formats for documenting patterns include the *Alexandrian* [35], *POSA*-style [34], and *GoF*-style [36]. Over time, and across different classes of systems, several patterns have been discovered and collected in *pattern catalogues*. Pattern catalogues are important, for they introduce a vocabulary that designers may leverage to communicate efficiently about designs. Examples of pattern catalogues include those on object-oriented design [36], concurrent systems [37], enterprise integration [38], messaging systems [39], reactive systems [40], fault-tolerant systems [41], embedded systems [42], etc. Catalogues typically present patterns in logical groups, or *categories*, that help to identify and relate patterns addressing similar problems. Various schemes for classification, or *taxonomies*, may be consider to show structure in pattern catalogues. For instance, object-oriented design patterns may be distinguished by type (creational, structural, behavioural) or scope (objects, classes).

2.2. Design patterns for the IoT and related scenarios

In this section, we briefly review pattern catalogues for the IoT, cyber–physical systems (CPSs), edge computing, multi-agent systems, and other related fields.

In [43], a small catalogue of five IoT design patterns is provided. Each pattern description includes a name, aliases, a graphical icon, a description of the problem, context, forces, solution, result, variants, related patterns, and an example. The *Device Gateway* pattern solves the integration problem among devices where some of them may not support the network communication protocol/technology. The *Device Shadow* pattern can be used to support interaction with devices intermittently going offline. The *Rules Engine* pattern is used to flexibly specify the behaviour of a system in reaction to messages or events. The *Device Wakeup Trigger* aims to support immediate communication with devices in sleep mode. The *Remote Lock and Wipe* pattern provides a secure solution to deal with stolen or lost devices.

In [44], a review of design patterns for decentralised self-adaptive IoT software systems is provided. The authors present 55 patterns in a taxonomy, inspired by the *Monitor-Analyse-Plan-Execute with Knowledge (MAPE-K)* control loop, as follows. Category *Monitor* includes patterns addressing monitoring problems such as deciding what to monitor, collecting data, and preprocessing data. Category *Analyse* includes patterns addressing the problem of data analysis with the intent of identifying opportunities and needs for adaptation. Category *Plan* includes patterns aiming at solving the problem of generating adaptation plans. Category *Execute* comprises patterns whose intent is to carry out pre-defined adaptation plans. Category *Component Structure and Interaction* deals

with the distribution of adaptation logic across the system component. Finally, category *Knowledge Management*, deals with the management of shared knowledge.

In [45], a systematic literature review of literature on architectural and design patterns for IoT systems is provided. They classify IoT patterns by abstraction level (architectural style, architecture, design), domain specificity (general non-IoT, general IoT, and domain-specific IoT), and addressed quality attributes (performance, compatibility, usability, reliability, security, maintainability, portability, scalability, privacy). Their analysis shows, among various things, that the proposed patterns often appear in a single paper, which means that there is limited consensus on IoT patterns.

Other relevant examples of pattern catalogues include those on industrial IoT [46], biologically-inspired coordination [23,24], field-based coordination [25], decentralised coordination [47], and decentralised self-adaptation with MAPE-K loops [48].

Sithole and Marshall [49] carried out an investigation about systematic ways to organise IoT patterns. They observe that IoT patterns are often organised by architectural styles (e.g., granularity, context, functionality), relationship with other patterns (e.g., by similarity, association, or complementarity), common attributes/semantics (cf. taxonomies, thesauri, ontologies), and design level (local, regional, and global). In this work, we are concerned with patterns that (i) follow a logically peer-to-peer architectural style (but whose deployment can target multiple kinds of infrastructures [50]), (ii) are coarse-grained in granularity, (iii) address the context of self-organisation and macro-programming, (iv) support collective coordination and behaviour as functionality, and (v) provide a way to connect the local level with the regional/global level of behaviour.

2.3. Pattern catalogues for collective behaviour

This section reports the existing works that, to the best of our knowledge, are more related to this manuscript: pattern catalogues of collective behaviour.

The most related work is by Fernandez-Marquez et al. [23] (which somewhat subsumes the catalogue by Nagpal in [26]). There, a catalogue of bio-inspired self-organisation patterns is provided. The catalogue consists of four basic patterns (repulsion, evaporation, aggregation, spreading), three composed patterns (digital pheromone, gradients, gossip), and five high-level patterns (flocking, foraging, quorum sensing, chemotaxis, morphogenesis). Some of these patterns (e.g., aggregation, spreading, gradients, gossip) have also been used in the patterns described in this work. This paper, instead, focusses on concurrent collective processes, and adopts a different perspective for what concerns the classification of the patterns (i.e., it is based on functional concerns rather than levels of abstraction).

Oh et al. [30] provide a review of self-organising multi-robot pattern formation. This is related since collective processes can also be used to denote dynamic groups of devices which, being displaced in space–time, correspond to pattern formations. However, the focus is very different, and not related to the software engineering and programming perspectives. In particular, the survey discusses both macro-level and micro-level models, and mentions mechanisms for shape formation like leader-follower structures, virtual structures, potential fields, and consensus-based control.

A notable related survey is also by Brambilla et al. [18], which broadly focus on the engineering of swarm robotic systems. They classify collective behaviours into four main classes: (i) spatially-organising behaviours (including pattern formation, morphogenesis), (ii) navigation behaviours (including collective exploration, collective transport, and coordinated motion), (iii) collective decision-making (including task allocation and consensus achievement), and (iv) others (including group size regulation and human-swarm interaction).

A fundamental difference w.r.t. the aforementioned related pattern catalogues is that they do not generally consider the programming perspective. In [23], the patterns are expressed as transition rules resembling chemical reactions working over patterns of tuples. [18,30] span different kinds of approaches (e.g., automated and behaviour-based), but do not show any code.

Regarding possible taxonomies of collective behaviour, Wood and Galton [51] propose to classify collective phenomena based on five criteria: (i) membership (constant or variable), (ii) location (none, fixed, or variable), (iii) coherence (due to cause or due to purpose), (iv) roles (members differentiated by roles or not), (v) depth (with or without sub-collectives). Though theoretically interesting for *understanding* collective, however, it does not address the engineering dimension.

2.4. Motivation

This work draws motivation from the significance of programming approaches for collective adaptive systems (i.e., a specific paradigm for a subset of IoT problems), and the need of design pattern catalogues for distilling the knowledge of recurrent problem-solution pairs in these settings. We detail this motivation in the following, and contextually clarify the positioning of this work.

Collective and decentralised computing. Abowd in [5] refers to *collective computing* as the next revolution in computing after ubiquitous computing [52], whereby heterogeneous collectives of humans and artificial agents synergically cooperate to solve complex problems. This vision is promoted by the IoT, i.e., by the ability of interconnecting and addressing smart objects situated in physical space. Indeed, an IoT (sub-)system can be seen as a *collective* of smart objects or agents that collaborate to pursue joint goals and tasks. Examples of collective-level applications include coordinating a fleet of autonomous vehicles, performing a distributed situation recognition (by collecting and processing data from a large set of sensors [28,53]), managing resources and tasks as in volunteer computing [54,55], supporting self-organisation of the computing infrastructures (e.g., for geographically sparse edge-fog-cloud deployments [55,56]), or crowd-based services (e.g., for monitoring [57] or sensing [58]).

Table 1
Relationship with respect to related work. The first block of references includes the most related works (i.e., sharing programming models and/or techniques). (*) Legend. *Description level* can be: Natural Language (NL), Formal Language (FL), Programming Language (PL).

Related work	Scope	Description level (*)	Taxonomy and Patterns	Relationship to this work
This Work	Decentralised collective processes (see Section 4)	PL	See Fig. 4	N/A
[25]	Self-stabilising aggregate computing patterns	PL	Patterns: Gradient-cast (information spreading), Collect-cast (information collection), Time-evolution (state tracking)	The patterns are self-stabilising implementations in the Protelis aggregate programming language of the patterns in [23].
[23], [24], [26]	Bio-inspired coordination patterns	FL	Basic patterns: repulsion, evaporation, aggregation, spreading. Composed patterns: gossip, gradients, digital pheromones. High-level patterns: flocking, quorum sensing, chemotaxis, morphogenesis.	Patterns are described in terms of chemical reaction-like transition rules. These patterns can be components of our collective patterns.
[22]	Spatial computing *domain-specific languages (DSLs)/approaches*	PL	Classes of space–time (S/T) operations: (i) measure S/T, (ii) manipulate S/T, (iii) compute over S/T, and (iv) evolving S/T. Classes of DSLs: (i) amorphous, (ii) biological, (iii) agent-based, (iv) sensor networks, (v) pervasive computing, (vi) swarms and modular robotics, (vii) parallel and reconfigurable computing, (viii) formal calculi.	Emphasis is on spatiotemporal abstractions/patterns. The survey broadly collects DSLs across domains and programming models.
[18]	*Approaches* for swarm behaviours	NL	(C1) Spatially-organising behaviours: aggregation, pattern formation, chain formation, morphogenesis, clustering. (C2) Navigation behaviours: collective exploration, collective transport, coordinated motion. (C3) Collective decision-making: task allocation, consensus achievement. (C4) Others: group size regulation, human-swarm interaction.	The catalogue collects methods for different collective behaviours/tasks in robot swarms.
[30]	Bio-inspired self-organisation *algorithms/models* for multi-robot pattern formation (PF)	NL/math	(C.1) Macroscopic PF: (C.1.1) Structured approach: leader-follower, virtual structure; (C.1.2) Behavioural approach: potential fields, consensus-based control, temporal delays. (C.2) Multicellular-inspired PF: morphogen diffusion, reaction–diffusion, gene regulatory network, chemotaxis.	Emphasis is on pattern formation and on algorithms/models rather than actual design/coding patterns.

A key research problem is the specification (i.e., defining *what*) and the execution (i.e., defining *how*) of the desired *collective behaviour*. The problem lies in how to define and reach a global goal in terms of the processing and interaction of the underlying set of computing devices. In general terms, the collective behaviour of the system may be defined through *automatic* means (e.g., based on machine learning techniques [59–61]) or *manual* means (e.g., based on specification or programming languages [17,19,62]). In this work, we focus on manual approaches based on *programming languages*.

Then, in general, two main approaches may be followed to organise the execution of a collective behaviour: an *orchestrational* approach where a centralised controller dispatches commands to the agents, or a *choreographical* approach where each agent knows the part it has to play.

A second crucial aspect is the system *topology*, namely the structure of the interaction/communication links connecting the agents. In this work, we focus on *scalability* and *resilience*, and so the idea is to avoid centralisations and global connectivity assumptions, which may act as bottlenecks or single points of failure. Therefore, this work follows a *choreographical*[1] approach where control is

[1] However, unlike classical choreographies, the computational model adopted in this work considers more homogeneous systems where roles are implicit and dynamic, emphasising self-organisation.

decentralised and interaction is local—i.e., where any agent interacts with a local portion of the environment and with a subset of other agents known as its *neighbourhood*.

Macroprogramming patterns. It turns out that the collection of entities, capable of communication and computation, of an IoT system can be abstracted as a single "distributed computer", which may be amenable to programming by a *global* or *macro-level* perspective. This idea has motivated research on *macroprogramming* [20], which originated in the context of wireless sensor networks [63] to later be applied to robotics and IoT-like systems [20,21]. To date, there is no catalogue of macroprogramming patterns. There do exist small catalogues of patterns for collective behaviour: e.g., in the context of spatial computing [22], biologically-inspired coordination [23,24], and field-based coordination [25]. In this work, we complement such contributions with a catalogue of patterns for collective processes. Described in detail in Section 3 according to the formalisation in [6], a collective process is a collaborative decentralised activity with a dynamic domain (i.e., running on an evolving set of devices), exploited with different solution schemas (i.e. patterns) in a variety of applications like client allocation to servers in content delivery networks [10], swarm control [12], sensing-driven clustering [14], spatial coordination [15], and smart societies [17]. In this manuscript, and specifically in Section 5, we will provide a pattern catalogue capturing existing and novel uses of this abstraction.

The relationship and differences w.r.t. related work (pattern catalogues) is summarised in Table 1.

3. Background: Aggregate programming for decentralised collective processes

This section describes the aggregate computing paradigm [62], which provides an *operational model* for understanding and implementing the patterns described in Section 5. In particular, Section 3.1 clarifies the system model and the execution model (i.e., a computational model for collective processes); Section 3.2 introduces the programming model, with examples in the ScaFi language [32] (i.e., a language approach for programming a single collective process); and Section 3.3 describes *aggregate processes* [6] (i.e., an abstraction for programming multiple decentralised collective processes).

3.1. Aggregate computing model

We adopt the aggregate computing model [62] as a general model for decentralised collective computing systems and collaborative tasks.

3.1.1. System model

In this approach, a *system* is a collection of interacting computing *nodes* (also called *devices*). A device has *sensors* and *actuators* that provide a generic interface for perceiving and acting upon the local environment/context. A device can only directly interact, by exchanging messages, with a subset of other devices—also called its *neighbourhood*. Indirect interaction is also possible, through the mediation of the environment (cf. stigmergy).

3.1.2. Execution model

The logical execution model is based on *asynchronous rounds* atomically consisting in *sense-compute-interact* steps. Rounds may be scheduled according to different policies. A simple policy is to schedule a round once every k milliseconds—the frequency will depend on the rate of change in the environment or the desired rate of actuation control. Among rounds, a device *sleeps*, but may still receive messages from neighbours. In detail, a round consists of the following steps:

- *Sense*. In this step, the device gathers all the relevant contextual information, i.e., sensor data and messages from neighbours. We assume that only the most recent message is kept per neighbour, and that messages *expire* after a certain (configurable) period of time (*retention time*).
- *Compute*. In this step, the device runs its control program, which maps the contextual information gathered in the previous step to an *output* that describes the actuations to perform and the messages to be sent to neighbours.
- *Interact*. In this step, the device runs the actuations determined in the previous step, and sends the prescribed messages to neighbours.

We stress that the system and execution model is *logical* and *general*. By logical, we mean that it abstracts from physical deployment details, such as the way in which neighbours are discovered, or the location at which the control program resides (e.g., it might be at the edge or in the cloud). By general, we mean that it does not fix detailed aspects of the execution: aspects like the scheduling of rounds, the retention time for messages, and the concrete sensing and actuation logic, can be defined on a per-application basis. Indeed, on-going research is carried out to define ways to optimise these aspects, e.g. using simulation and/or learning.

3.2. Aggregate programming model

In aggregate programming, the developer writes a *single program* (also called the *aggregate program*) which is logically executed *fully* by all the devices in the system at their *compute* steps. The programming model is formalised by *field calculi*, a family of core languages providing primitives for manipulating computational fields. A *computational field* is a function (or map) from a domain of devices to a codomain of computational values. For instance, a field may capture the temperature sensor readings of the devices located at a certain area, or the distances between a device and its neighbours. Field calculi are implemented by concrete aggregate programming languages such as ScaFi, a Scala-internal DSL. In the following, for the sake of self-containment of this manuscript, we provide a brief introduction to aggregate programming with ScaFi.

Local expressions. A local expression such as `1+3` or a built-in function call like `mid()` to provide the running device's identifier, or `sensor[Double]("temperature")` to read a double-precision floating-point value from the temperature sensor, is evaluated as usual and yields the result value. If we consider that expression evaluated by a system of devices, then we can think at its result as a field of values—a constant, uniform field of 4s; a constant, non-uniform field of device identifiers; or a generally non-constant, non-uniform field of temperature readings (recall that programs, and hence expressions, are evaluated round by round).

Stateful evolution. An expression

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

is evaluated by running, in each round, the function provided as second argument while binding its argument `x` to the value computed the previous round (or to the initial value `0`, in this case, on the first round, as provided by the first argument of `rep`). The second argument of `rep`, in other words, is a function that computes a new result in terms of a previous result `x`. The example at hand just increments `x`, starting from zero: so, the overall expression effectively counts the number of rounds executed by a device. Notice that if different devices schedule rounds at different frequencies, those devices will generally exhibit different values for that expression for an observer able to observe them roughly at the same instant. In a nutshell, `rep` provides a way to store and evolve *local state*: the newly computed result will be conceptually stored in the device state, and made available at the next round.

Neighbourhood interaction. The expression:

```
// Signatures:
// def foldhood[T](init: T)(f: (T,T)=>T)(e: => T): T
// def nbr[T](e: => T): T
foldhood(Set.empty[ID])((acc,v) => acc.union(v)){ nbr{ Set(mid()) } }
```

yields, in any device, the set of IDs of its neighbours (including self, since we assume a device is a neighbour of itself). Basically, `foldhood` is the classical fold operation of functional programming: it accepts an initial value (in the example, the empty set), a binary operator to merge values, and the collection to fold over is a set of values—one per neighbour. The collection of neighbouring values is obtained by evaluating the expression provided as third argument to `foldhood`, by substituting `nbr(e)` expressions with the value of expression `e` calculated by the neighbour. In other words, `nbr(e)` expressions support bi-directional communication: the locally evaluated value for `e` will be shared with neighbours, and neighbours will share their value. In the example, each device shares the singleton set with its ID; so, folding over such a collection of values from neighbours will allow a device to collect all the neighbour IDs.

Combining state evolution and neighbourhood interaction: the gradient. The basic technique for letting information propagate beyond neighbourhoods and eventually taking global effects is to combine `rep` and `foldhood`/`nbr`. The paradigmatic example is the *gradient*, a (type of) algorithm that yields the field of minimum path distances from source devices. It can be encoded as follows (also: notice how functions can be defined to encapsulate collective behaviours as functional blocks):

```
def gradient(source: Boolean): Double =
  rep(Double.PositiveInfinity)(g => {
    mux(source) { 0.0 } {
      foldhood(Double.PositiveInfinity)(Math.min) {
        nbr{g} + nbrRange()
      }
    }
  })
```

where `mux(c,e1,e2)` evaluates `e1` and `e2` and returns the value of the former if `c` evaluates to `true` or the value of the latter otherwise; and `nbrRange()` is a *neighbouring sensor* (provided by the platform) that behaves like a `nbr` and shares, for each neighbour, an estimation of the distance to the executing device. This basic implementation works as follows: the source device returns `0.0` (and will share such value with its neighbours), whereas the other devices will return the minimum value among those obtained by summing neighbours' gradient values with the corresponding local distances. Notice that a gradient field is a structure able to support efficient *information flows*, supporting propagation (cf. gossip) and collection (cf. distributed summarisation) of information across a distributed network of devices, and is therefore a fundamental pattern of self-organisation upon which more complex patterns can be built.

Computation branching. Suppose we have a system we two kinds of devices (red and blue), and that we want at times to consider them altogether or separately (i.e. to possibly execute computations separately in each domain). Consider:

```
val g1 = gradient(source1)
val g2 = branch(isRed) { gradient(source2) } { gradient(source3) }
```

Gradient `g1` is computed by considering all the devices, whereas `g2` holds the gradient computed in the red subnetwork (for red devices) or in the blue subnetwork (for blue devices). Notice that a red device may have a blue device as neighbour, but inside the two branches, the folding of the gradient block will only consider as neighbours the device that executed the same branch.

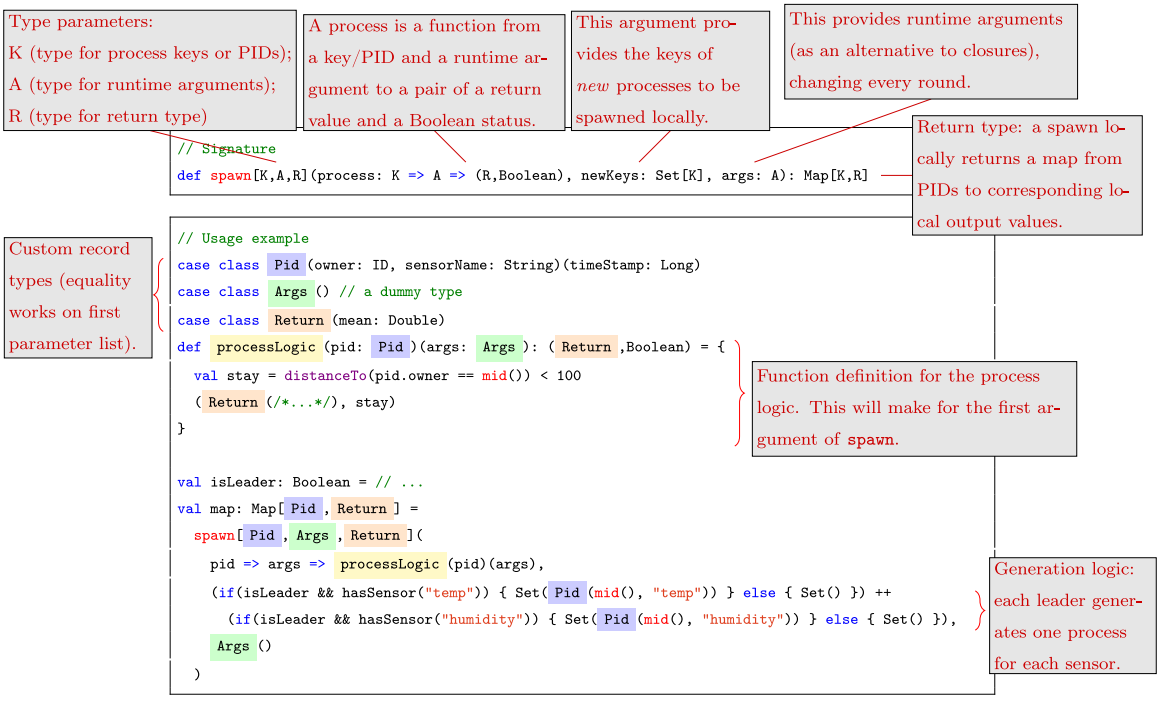


Fig. 1. Definition and example usage of the `spawn` function, which is the construct that is responsible of launching and running instances of a process type.

3.3. Aggregate processes

An *aggregate process* [6,7,64] can be seen as an independent aggregate computation associated with a *process identifier (PID)* (also called a *process key*). Technically, we should distinguish an aggregate process *type*, which defines an aggregate computation (like a class in OOP), from an aggregate process *instance*, which is a running copy of a process type (like an object in OOP). For convenience, we will sometimes use shorthands in terminology, and use term "aggregate process" (or also "collective process", or simply "process") to refer to an aggregate process type or to an aggregate process instance, whenever the right acceptation will be clear from the context.

By a programming language perspective, aggregate process types are defined as *binary functions* of two inputs (a PID and an argument) producing *pairs of values* (of a *result* and a *status*), i.e., of a Scala type `K=>A=>(R,Boolean)` (for generic types `K`, `A`, and `R`):

- *First input:* the PID, which identifies the process instance and also embeds *construction arguments* (i.e., fixed at the time in which a process is instantiated).
- *Second input:* a *runtime argument*, if needed (e.g., an expression evaluating to values that may possibly change from round to round).
- *Output:* it is a pair of a *result* of an arbitrary type and a *Boolean status* denoting whether the node is part of the process instance or not. The status is key for the dynamics of the process:
  - *Status is* `true`: the node is part of the process instance, it will interact with other nodes participating to the same process instance, it will exhibit the result of the process instance execution, and, crucially, it will automatically distribute the PID to all its neighbours (currently participating in the process instance *or not*).
  - *Status is* `false`: the node is *not* part of the process instance, it will *not* interact with other nodes participating to the same process instance, it will *not* exhibit the result of the process instance execution, and, crucially, it will *not* automatically distribute the PID to any of its neighbours.

As an example, consider the custom datatypes and function `processLogic` in Fig. 1.

For a process defined as above, the generation of instances and their execution can be handled through a call to a generic function `spawn[K,A,R](processLogic,keys,args):Map[K,R]`, formalised in [64] explained in Fig. 1 (top). In particular, a call to `spawn` on a given round is evaluated as follows:

- its three arguments are evaluated: we can assume `processLogic` is assumed to be static/constant; argument `keys` provides a set of PIDs corresponding to (possibly new) process instances to be executed; and argument `args` holds a value to be passed to all the running instances of the process;

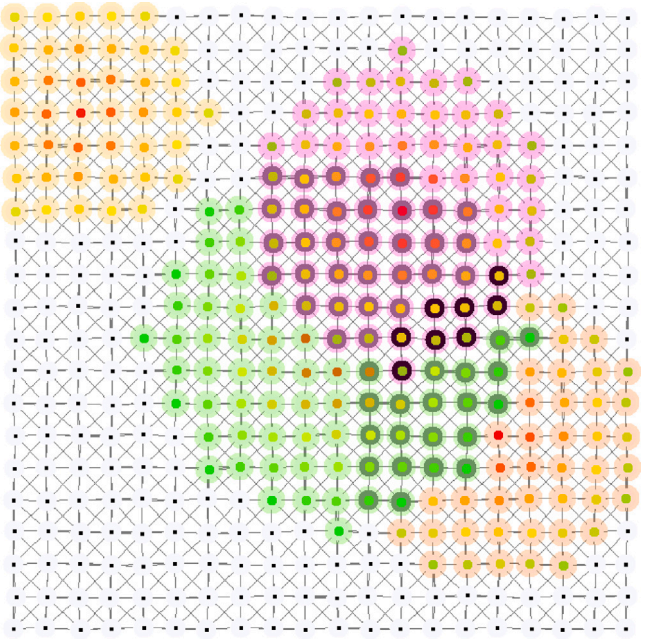


Fig. 2. Snapshot of a simulation executing the multi-gradient example. The nodes running processes are denoted by three coloured circles: the largest circle denotes the process instance (i.e., different colour for different PID); the medium circle is darker if more instances are concurrently run; and the smallest circle denotes gradient values (from red at the source to greener values). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

- three possibly overlapping sets of process instances will be executed:
  1. *previously played instances*, executed by the same device at its previous round and for which the output status was `true`;
  2. *shared instances*, namely those corresponding to PIDs shared by neighbours;
  3. *new instances*, namely those corresponding to PIDs provided by argument `keys`.
- the output will be a map from PIDs to corresponding results (instances for which status was `false` are filtered out).

Notice that if a node quits a process instance in a round, it may still play the same instance the next round, if any neighbour shared it—however, in this case, the node will execute the process with a new starting state (e.g., `rep`s will be resetted).

3.3.1. *Example: spatially limited multi-gradient*

One problem with the gradient algorithm shown before is that if there are more sources in the system, each device will only return the distance to the closest source. We could have multiple calls to `gradient` against different sources, but how if the source set is dynamic? Additionally, sometimes we would like to run the gradient only in a limited area of the system, to save resources from devices that are very far away. A solution is to use `spawn` to generate an independent and spatially limited gradient computation for each source: we call this a *spatially limited multi-gradient* (see Fig. 2). The example can be programmed as follows:

```
case class Pid(source: ID)(val size: Double)
def logic(pid: Pid)(source: Boolean): (Double,Boolean) = {
  val g = gradient(pid.source == mid() && source)
  (g, g <= pid.size)
}
val pids = if(isSource) Set(Pid(mid())(SIZE)) else Set.empty
val gradients = spawn(logic, pids, isSource)
```

where `isSource` and `SIZE` are assumed to be provided. The process' logic computes a `gradient` from the process' `source` (stored in the PID): the return value is a pair of the gradient value and a status which is `true` (internal) iff the gradient value is less than the process' `size`. Notice that when `isSource` becomes false, the gradient value will become to raise, eventually causing all the devices to quit the process instance.

4. Method: Pattern identification, classification, and description

The collective process abstraction introduced in Section 3 can serve as a building block for the design and implementation of certain kinds of collective IoT systems—especially those involving self-organisation and decentralised collaboration within large

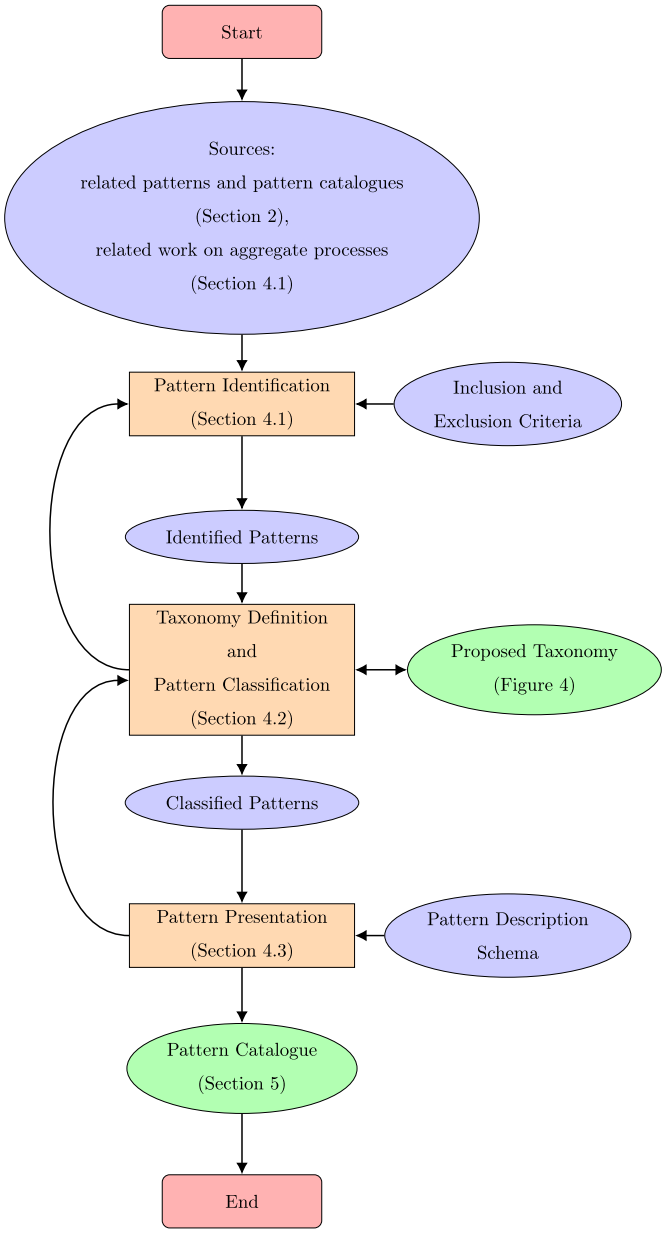


Fig. 3. Summary of the process described in Section 4. Ellipses denote data/artefacts, whereas squares denote sub-processes (in green, the main contributions).. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

dynamic networks of devices. Its implementation in aggregate computing, the so-called "aggregate process", paves the way to a pragmatic *programming* dimension, where recurrent solutions or patterns take form. After several implementations and case studies leveraging collective process abstractions (e.g. [4,6,15,31]), and taking into account related research (cf. Section 2), a pattern catalogue can be delineated. This effort is motivated by three main benefits: (i) extending existing catalogues of IoT patterns with other complementary patterns covering the *collective dimension* of IoT systems [5]; (ii) providing a higher-level account of unconventional programming paradigms like aggregate computing [62]; and (iii) giving structure to previous research to collective adaptive computing [19].

Given the importance of proper pattern organisation for the IoT [49], in this section we describe the method adopted for identifying (Section 4.1), classifying (Section 4.2), and presenting (Section 4.3) the patterns in the catalogue exposed in Section 5. The overall process is summarised in Fig. 3.

4.1. Pattern identification

Sources. Patterns are not invented, but discovered from instances of repeated adoption. As sources supporting identification of collective process patterns, we referred to related pattern catalogues (cf. Section 2.3), previous work adopting aggregate processes as a component of their contribution [6,7,14,31,64–67], and related work on self-organisation and collective computing [19,25–27,29,68]. We considered primary studies published in workshops, conferences, and journals, and where the solutions are expressed using formal methods (e.g., DSLs) or pseudo-code.

Identification method. An iterative approach has been followed, combining bottom-up analysis of solutions and source code with top-down analysis driven by the emerging classes of solutions (see Section 4.2). Also, the possibility of "dual", symmetric formulations has been contemplated: for instance, spatial and time aspects can often be considered orthogonally, and mobility can be considered by the point of view of devices (moving in some environment) or of the processes they host (moving across the devices). The identification has been implemented as a multi-phase process where candidate patterns have been (i) creatively discovered at first as mentioned above, (ii) implemented and tested on synthetic scenarios to qualitatively validate the ability to fulfil their intent, and (iii) preliminarily described (as discussed in Section 4.3) and classified (cf. Section 4.2) in order to synthesise the key characteristics of the pattern in a structured form (e.g., code schema, related patterns). The description of each candidate pattern and the corresponding tentative classification have been used to validate the candidate patterns into the final proposed patterns.

Inclusion and exclusion criteria. In this work, we focus on patterns of *collective processes*, regulated through the mechanisms described in Section 3 and specifically in Section 3.3. Accordingly, we included patterns for large-scale decentralised coordination and self-organisation, but excluded other interactive patterns aimed at collective intelligence but based on orchestration or multi-party protocols (which generally focus on more small-scale and heterogeneous systems, i.e., multiple parties playing different roles). Moreover, we have excluded more fundamental and already well-known patterns of collective behaviour (such as information flows [27], gradients [26], distributed summarisation [28], leader election [29], digital pheromones [12,23]), which have been extensively covered in previous work. Indeed, several of these more fundamental patterns are *used* as components by the proposed patterns of collective processes.

4.2. Pattern classification: Proposed taxonomy

As mentioned in Section 2.1, it is common to organise patterns into coherent groups, as supported by some *taxonomy* (classification scheme). This is useful to provide a structure for collections of patterns (cf. pattern catalogues) and to help relating patterns by various criteria (like goals, approach, or assumptions). For instance, the GOF patterns were classified in three classes: creational, structural, and behavioural.

Similarly, commonalities can be found across the collective process patterns proposed in this work, to group them into coherent clusters. Considering the `spawn` mechanism introduced in Section 3.3, and its dynamic behaviour, two main aspects immediately emerge: the *domain dynamics* of collective processes across the network (regulated through their *status* field), and the *collaborative behaviour* internal to a given collective process instance (promoting a certain *output*). Moreover, the aspect of domain dynamics can be further distinguishes into three main phases: *generation* (creation of one or more collective process instances), *evolution* (ensuring that the domain evolves to support the desired activity), *destruction* (ensuring that the process instance is removed from the system).

Based on the above considerations, the proposed classification of patterns of decentralised collective processes is as follows (see also Fig. 4):

- **L.** *Lifecycle Management (Creational and Destructional)*: these patterns focus on the *lifecycle* of process instances, namely on their creation and destruction.
- **B.** *Behavioural (Intra-Process Behaviour)*: these patterns focus on the activity internal to collective process instances, e.g., to support the coordination of the participants.
- **M.** *Mobility*: these patterns address aspects related to the movement of devices and processes across the devices.

Notice that the mobility aspects are very related to the domain dynamics of processes and hence to lifecycle management; still, they are grouped in a distinct category since, in that case, creation and destruction are more intertwined.

Finally, aspects related to the *interaction* of different collective processes (cf. *control-flow* perspectives as in workflows [69]) are not considered, for these are still to be addressed by future research.

4.3. Pattern description schema

To describe each pattern, we propose a lightweight description schema consisting of the following components:

- *Name.* It provides a name useful to succinctly characterise the pattern (e.g., by capturing its intent and solution idea) and to clearly refer to it.
- *Aliases.* This section includes synonyms, alternative names of the pattern that may have been taken from previous work or that may emphasise other key aspects of the pattern. Together with the *Name*, this contributes to the definition of a *vocabulary* of collective process patterns.
- *Intent.* This section concisely describes what the goal of the pattern is (in terms of the problem addressed and the core idea).

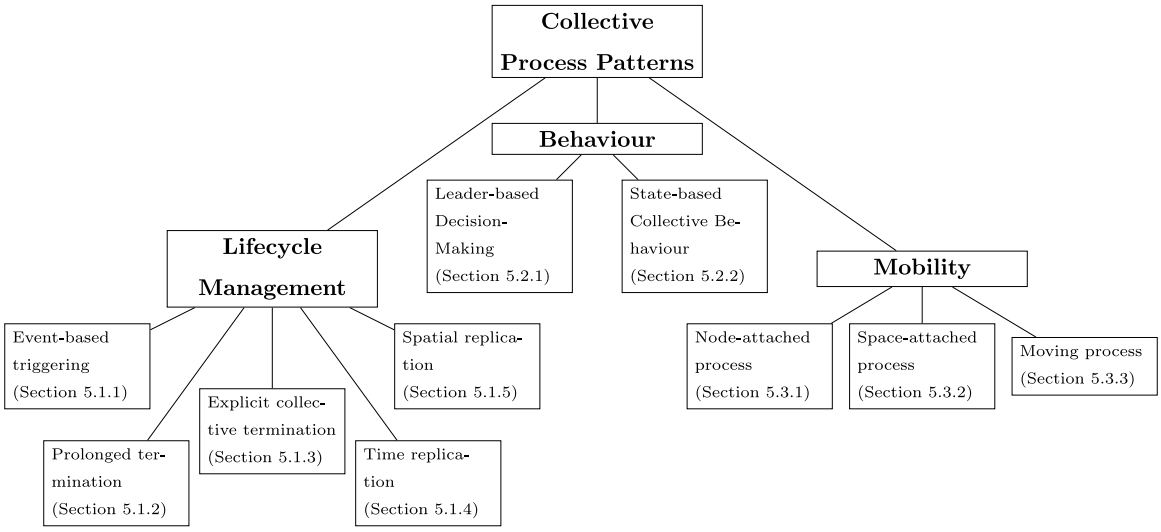


Fig. 4. A taxonomy of the patterns of collective processes covered in this work.

- *Context and Example(s).* This section mentions the general settings and example scenarios in which the pattern can be useful and/or applicable as well as *known uses* of the pattern from the literature.
- *Description.* This section describes the solution proposed by the pattern in a rather implementation-agnostic way, by illustrating its key components and how they interact.
- *Implementation.* This section provides one or more implementation schemas for the pattern, using ScaFi code (cf. Section 3).
- *Discussion.* This section provides clarifications and discusses assumptions, implications, and other aspects related to the pattern, its solution, and its implementation variants. Moreover, it discusses similarities and differences with respect to other patterns.

Notice that these components of the pattern description are similar to those found *Alexandrian* [35], *GoF* [36], and POSA formats [34], where some of their components are aggregated for a more effective presentation. For instance, the distinction in the GOF pattern description between *Structure*, *Participants*, and *Collaborations* (and, similarly, the POSA distinction between *Structure* and *Dynamics*) have been condensed into a single *Description* section because, in collective process patterns, structure and dynamics tend to be more intertwined.

5. A catalogue of patterns for decentralised collective processes

The proposed catalogue of patterns for decentralised collective processes is illustrated in Fig. 4, and organised as per the taxonomy presented in Section 4.2. This section is also structured following the taxonomy, with a subsection per class. In each subsection, the patterns are presented following the pattern description schema defined in Section 4.3.

As summarised in Fig. 4, the following patterns are presented:

L. *Lifecycle Management (Creational and Destructional)*: these patterns focus on the creation and destruction of process instances.

(a) Event-based triggering (Section 5.1.1)
(b) Prolonged termination (Section 5.1.2)
(c) Explicit collective termination (Section 5.1.3)
(d) Time replication (Section 5.1.4)
(e) Spatial replication (Section 5.1.5)

B. *Behavioural (Intra-Process Behaviour)*: these patterns focus on the activity carried out within one process instance.

(a) Leader-based decision-making (Section 5.2.1)
(b) State-based collective behaviour (Section 5.2.2)

M. *Mobility*: these patterns relate collective activities with the local activity of mobile agents.

(a) Node-attached process (Section 5.3.1)
(b) Space-attached process (Section 5.3.2)
(c) Moving process (Section 5.3.3)

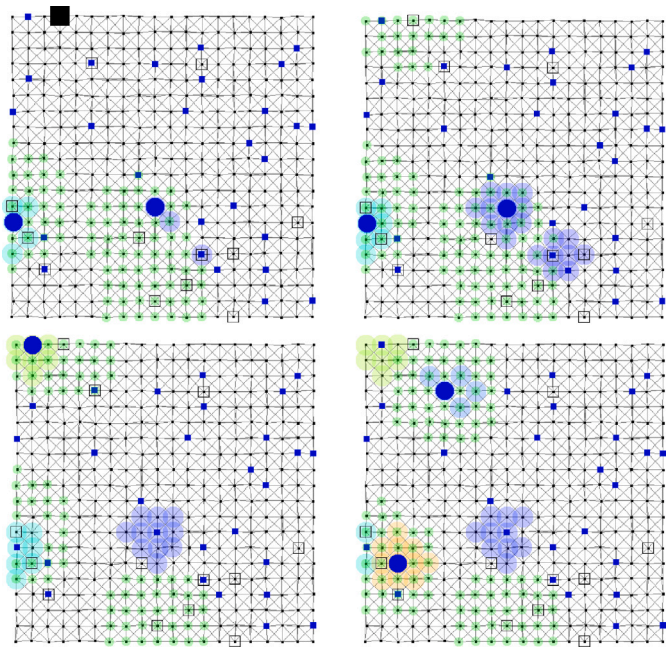


Fig. 5. Screenshots from a simulation of a system based on event-based triggering. Notation: small black dots denote devices; edges denote neighbouring relationships; event detectors are denoted as empty squares, which are filled with black during the round in which they have detected an event; event handlers are denoted as small blue squares, with a large blue oval superimposed appearing only when the device is currently handling an event; reporting is denoted by small green shadows; the larger semi-transparent coloured shadows denote handling process instances (different colours for different instances). The screenshots show consecutive moments of the simulation: (top-left) two handling processes are active, and a device at the top detects an events; (top-right) a new reporting process is started; (bottom-left) the reported event is handled by the closest event handler; (bottom-right) two new handling processes start, while previous handling processes (at the top-left of the arena and near the centre) are being closed. Notice that the reporting process initiated by the detector at the bottom-centre of the arena does not reach any handler. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

We cover lifecycle management patterns first, for they address the most basic aspects, namely creation and destruction of collective process instances.

As convention, in the code snippets, we will use abstracted functions (denoted as `someFunction`), comments, or the valid Scala keyword `???` – three question marks – to denote arbitrary portions of code. Moreover, we will assume certain abstract functions are defined, to capture logic that is needed for a full example but that is not particularly relevant for the example at hand:

```
case class Pid(pid: Long) // basic type for PIDs
case class Args() // dummy type for empty runtime arguments
case class Return() // dummy type for return values

def pids(): Set[Pid] // generation logic
def statusLogic(): Boolean // process status logic
def outputLogic(): Boolean // process output logic
def processLogic(pid: Pid)(args: Args): (Return,Boolean)
```

5.1. Lifecycle management patterns

Lifecycle management patterns focus on the creation and destruction of collective processes.

5.1.1. Event-based triggering

Name. *Event-based Triggering*

Aliases. *Observer*, *Detector-Handler*

Intent. The intent is to support the flexible generation of collective process instances upon the occurrence of events of interest.

Context and example(s).

- *Surveillance*. An object enters an area. One or more devices at the border detect it is a suspicious object, and issue a *k-coverage* [9] collective process to ensure that the object is monitored by at least k monitor devices (e.g., cameras) in the surveillance area.

- *Rescue*. A swarm explores a partially unknown environment to detect injured people. A person is detected. The leader decides to split the swarm into two groups: one to rescue the person back to the base, and the other to continue the exploration.
- *Smart city management*. A set of sensors in a contiguous geographical area report what waste containers are nearly full. The area manager detects that an over-threshold number of waste containers require collection, and starts a collecting process.

Description. An *event detector* device identifies an event of interest (e.g., a dangerous situation) and communicates the corresponding event description to one or more *event handler* nodes (typically the closest one). Such a communication may be implemented through a spatially limited broadcast (possibly using a collective process in turn). An event handler acts as a process initiator: it locally spawns a collective *event handling* process. The event itself may be a purely local occurrence or the outcome of a collective process (e.g., a distributed estimation). Specific forms and variants include the following:

- *Basic form.* The simplest case is when the same node acts as event detector and event handler.
- *Exclusive handlers.* A variant is when each event has to be handled by at most (or exactly) one handling process. This can be achieved, e.g., by having the detector choose among potential handlers.
- *Multiple types of events.* A complex variant is when there are multiple types of events. In this case, event detectors and event handlers may be able to detect and handle a subset of the event types.

A graphical example of this pattern in simulated settings is provided in Fig. 5.

Implementation. It follows a basic implementation schema in ScaFi:

```
case class Event(/* ... */)
def eventHandlerProcess[T](e: Event)(args: Args): (T, Boolean)
def detectEvents(): Set[Event]
def isHandler(): Boolean

val events: Set[Event] = detectEvents()
val gradientToHandler = distanceTo(isHandler())
val eventsFlow: Set[Event] =
  C(gradientToHandler, _++_, events, Set.empty)
val eventsToBeHandled = if(isHandler) eventsFlow else Set.empty
val processInstances =
  spawn(eventHandlerProcess(_)(_), eventsToBeHandled, Args())
```

Essentially, the idea is that event detectors perceive local events (through an abstracted function `detectEvents()` based e.g. on local sensors) and these events are *collected* through a gradient to the closest handler device (where handlers are denoted by the abstracted Boolean field `isHandler`). The implementation leverages the "collect-cast" `C(g,aggr,data,none)` building block [28] which collects `data` into the sinks of gradient `g` by aggregating data using function `aggr` (in this case, `_++_` refers to the union of sets), with `none` as zero-value for aggregation (provided by neighbours for which the current node is not a parent in the gradient-induced spanning tree). Finally, only the handlers will leverage the collected events (cf. `eventsToBeHandled`) for starting corresponding collective process instances.

Discussion. It is important to note how the set of events accumulated into the handler devices are the result of a collective computation. In the provided implementation schema, the events are just disseminated from event detectors; however, the aggregation logic could be more complex, combining events to generate new ones. Alternatively, the field of events to feed the spawning of processes, `eventsToBeHandled`, could be determined by a complex collective and spatiotemporal computation, e.g., something as "in a crowd management scenario, start a dispersion process once there is a severe crowding for more than one minute, and there is a limited number of narrow escape paths" [70]. Another aspect that may be addressed in implementations is the relationship between events of the same or different types: e.g., more recent events may subsume others, or more priority events may cause the stopping of others (this has to be achieved through a proper process termination logic in `eventHandlerProcess`).

5.1.2. Prolonged termination

Name. *Prolonged Termination*

Aliases. *Conclusive Termination, Once-for-all Exit, Definitive Quitting, Re-entrance Avoidance*

Intent. The intent is to let a device quit a process instance for a prolonged period (or even once for all), i.e., with the guarantee that there will be no re-entrance anytime soon.

Context and example(s). A device may definitely lose interest in (or the right of) participating in a process. For instance, in the context of volunteer computing [71], the owner of a device chooses to stop offering resources. Or, in a cooperative task, a node is distrusted [72], and made to quit the process.

Description. The problem addressed by Prolonged Termination is that of *re-entrance*, namely the possibility for a node to join again a process which had been left in the past. Re-entrance can be avoided in two ways:

- by avoiding a subsequent gathering of the PID of the quitted process (from neighbours or from local generation);
- if a subsequent gathering of the PID of the quitted process happens, the process should be quitted.

The pattern comes in two main variants:

1. participation to a process is suspended (i.e., termination is prolonged for a limited amount of time);
2. termination is conclusive.

Different techniques may be adopted to implement this pattern.

Implementation. The basic idea for an implementation is to have every device willing to quit a process keep track of the corresponding PID to prevent re-entrance in the future. For a non-final termination, each PID may be associated with a timestamp, and such a proscription may be cleared after a configurable period of time.

```
case class Return(terminate: Boolean = false /* .... */)

def processLogic(pid: Pid)(args: Args): (Return, Boolean) = {
  // ...
  val status = statusLogic()
  (Return(terminate=!status), status)
}

def proscriptionGC(ps: Set[Pid]): Set[Pid]

rep[Set[Pid]](Set.empty)(proscription => {
  val map = spawn[Pid, Args, Return](pid => args => {
    branch(proscription.contains(pid)) {
      (Return(true), false)
    }{
      processLogic(pid)(args)
    }
  }, pids(), Args())
  /* do something with the map of processes */
  proscriptionGC(proscription ++ map.filter(_._2.terminate).keySet)
})
```

In this implementation schema, the return value is used to express to willingness to quit the process, based on an arbitrary condition `wannaTerminate`. Then, we use the stateful construct `rep` to keep track of the PIDs to definitely quit into a `proscription` set. The actual `processLogic` is wrapped with a `branch` expression that avoids any computation and immediately quits the process if the PID belongs to the `proscription` set. The abstracted function `proscriptionGC()` is used to perform a *garbage collection* of proscriptions to avoid memory leaks (e.g., may clean out PIDs once enough time has passed.

Discussion. The crux of the problem addressed by this pattern lies in two key aspects of the dynamics of the notion of collective processes discussed in this paper: (i) the round- and repetition-based execution model (Section 3.1.2), and (ii) the implicit process propagation to neighbours carried out by the `spawn` construct, which is instrumental for the spreading and general evolution dynamics of collective process instances. Especially, the latter mechanism complicates process termination, since a device that quits a process instance may receive the same process instance immediately the next round if any of its neighbours are still executing it. Therefore, the process logic has to take into account the possibility of re-entrance. In the implementation schema, an explicit *proscription* set is leveraged to keep track of exited processes. It is important to clean proscriptions to avoid memory leaks, but the garbage collection policy should guarantee undesired re-entrance. Also, notice that this patterns allows the status logic to distinguish the willingness of a conclusive termination from a "normal" termination allowing re-entrance if appropriate for the process dynamics. Finally, re-entrance management could be supported by a collective computation (e.g., a device may decide to leave a process for good if it realises its area includes several more powerful devices), but the re-entrance decision is purely local. This pattern does not address actual termination logic: instead, the related pattern *Explicit Collective Termination (Section 5.1.3)* provides a simplified interface global process-wide termination.

5.1.3. Explicit collective termination

Name. *Explicit Collective Termination*

Aliases. *Event-based Collective Termination, Termination Signal, Global Shutdown*

Intent. The intent is to explicitly support the collective termination of a collective process instance, i.e., to ensure that all the participants quit the process.

Context and example(s). Once a collective process has reached its goal, every participating device can safely quit it. However, certain participating devices may be unaware that the goal has been accomplished. In this case, an indication that the process should terminate everywhere has to be communicated to all the participating devices. For instance, if the goal was to deliver a message to a recipient device through a hop-by-hop channel, the recipient could trigger the shutdown once it receives the message (cf. the "multi-hop chat example" in Fig. 12 for the *Moving Process (Section 5.3.3)* pattern). In the case of a task offloading process, the offloader can terminate the process once it gets the results (or an acknowledgment) back.

Description. The pattern establishes the possibility of terminating a collective process instance *globally* (i.e. for all the devices) by a *single, local termination signal* issued by an individual device. This way, the programmer does not need to think about complex termination logic, and just focus on the condition for termination to be initiated. Usually, the possibility of issuing termination signals is reserved to devices playing "special roles" in the process computation (e.g., the recipient of a message in the multi-hop chat example). The pattern may also define multiple *types of termination signals* to support different termination algorithms [73].

Implementation. It follows a basic implementation schema in ScaFi:

```
trait Status
case object ExternalStatus extends Status
case object BubbleStatus extends Status
case object TerminalStatus extends Status
final case class POut[T](result: T, status: Status)

def statusSpawn[K, A, R](process: K => A => POut[R],
                    params: Set[K],
                    args: A): Map[K,R] =
   spawn[K,A,Option[R]](k => a =>
     handleOutput(handleTermination(process(k)(a))),
     params,
     args
   ).collectValues { case Some(p) => p }

def handleTermination[T](out: POut[T]): POut[T] = {
  rep[(Boolean,Int,POut[T])]((false,0,out)){
    case (terminated,k,res) =>
      val mustTerminate = out.status==TerminalStatus |
        includingSelf.anyHood(nbr{terminated})
      val mustExit = includingSelf.everyHood(nbr{mustTerminate})
      (mustTerminate, 1,
        if(mustExit || (mustTerminate && k==0))
          (out.result, External)
        else out
      )
    }._3
  }

def processLogic[T,K](k: K)(args: Args): (Status, T) = {
  // ...
  val done = ??? // true when the overall process must quit
  if(done) (TerminalStatus, nullOutput)
  else (statusLogic(), outputLogic())
}

statusSpawn(processLogic(_)(_), pids(), Args())
```

where `statusSpawn` is a variant of `spawn`, possibly reusing the latter, that is able to deal with termination signals (cf. `TerminalStatus`). Basically, the termination logic is implemented in `handleTermination`: it works through a gossip of the termination signal, and taking precautions to avoid re-entrance (i.e., a process re-joining the process after quitting it). As seen for the *Prolonged Termination (Section 5.1.2)* pattern, a way to avoid re-entrance is to keep a set of the PIDs of quitted processes, and using it to filter received PIDs; this approach, however, needs a sort of garbage collection. Another issue is given by discontiguous process partitions: in this case, it is possible that only one partition gets to close the process. Dealing with this may require the adoption of other termination patterns, like Evaporation or Heart Beat.

Discussion. This pattern differs from *Prolonged Termination (Section 5.1.2)* since the latter promotes non-reentrant termination of *a single* device, whereas this one ensures that *all* the process participants quit (and provides a simplified interface in terms of a single local signal to be issued).

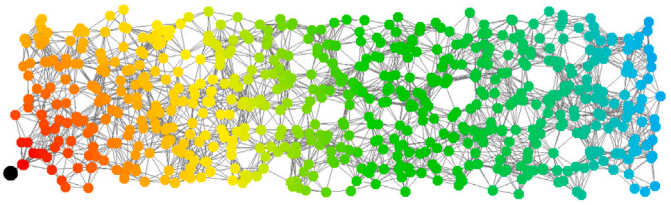
5.1.4. Time replication

Name. *Time Replication*

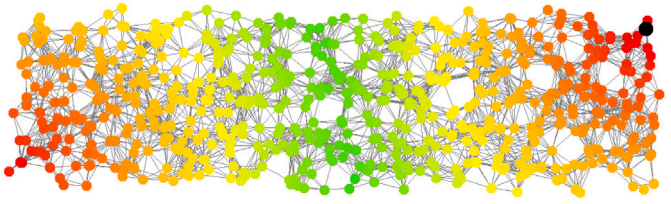
Aliases. *Temporal Spawn*, *Periodic Execution*

Intent. The intent is to support the instantiation of multiple collective process instances at different time instants.

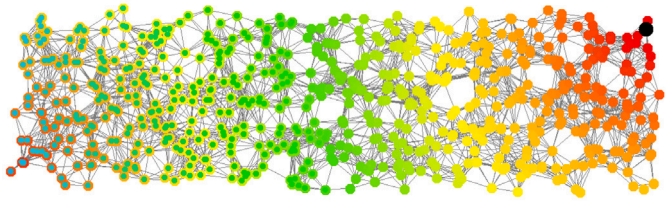
Context and example(s). Certain kinds of activities are periodic in nature. One example is garbage collection (e.g., in a smart city or in a computational ecosystem as in Section 5.1.3). Garbage collection may be used in the context of collective processes to clean up the maps of quitted PIDs (that serve to avoid re-entrance) to free memory (cf. the *Prolonged Termination (Section 5.1.2)* pattern). Another notable usage of time replication is to improve the dynamics of gossip [74]. Indeed, certain gossip algorithms like the gradient are fast to converge in one direction (decreasing values) but slow to adapt in the other direction (raising values); running delayed instances of gossip processes provides a way to exploit the fast convergence while re-starting from subsequent states of the environment. This is witnessed by the simulation results reported in Fig. 6.



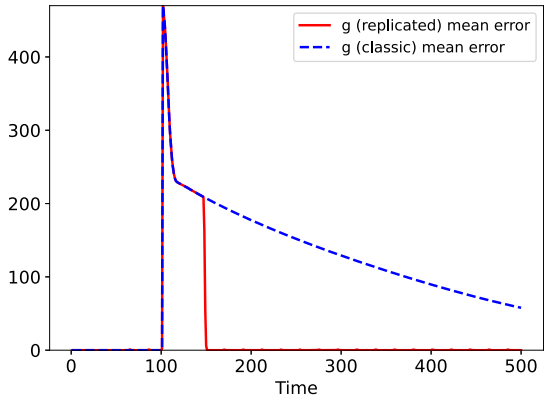
(a) The gradient from the source (black dot) has stabilised (converged to fixed values). Hot (resp. cold) colours denote low (resp. high) gradient values, i.e., short (resp. long) distances to the source.



(b) The source switches at the opposite part of the network: the stabilisation is fast for values that need to drop (in the right half) whereas it is slow for values the need to raise (in the left half).



(c) While the replicated gradient adjust quickly, the classical gradient takes a lot of time adjusting values (due to the raising value problem).



(d) Error plot (witnessing the aforementioned dynamics).

Fig. 6. Classic gradient versus replicated gradient. In the snapshots 6(a)–6(c), the black circle denotes the source of the gradient, whereas different colours denote different gradient values (hot colours for low values and cold colours for higher values). The inner circle denotes the replicated gradient's value, while the outer circle the classical gradient's value: the difference is only noticeable in the left part of the arena in Fig. 6(c). Finally, Fig. 6(d) shows how the replicated gradient enables faster convergence than the classic gradient (notice the quick drop in the error around $t = 145$). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Description. This pattern is basically a variation of pattern *Event-based Triggering (Section 5.1.1)* where the events are *time events*. Such time events can be purely local or distributed time (cf. logical clocks). Local time may leverage sensors for *local system clocks* or be expressed in terms of *number of rounds*. Distributed time could be expressed in terms of *causality* relationships [75,76]. So, a time event causes the generation of a new process instance, namely a *replica*. Besides time-staggered generation of replicas, the solution has to properly handle termination to keep a *limited number of replicas active at a certain time* (to avoid excessive resource consumption).

Implementation. Consider:

```
def clock(len: Long, decay: Long): Long =
  rep((0L,len)){ case (k,left) => // Function by pattern matching
    branch (left == 0){ (k+1,len) }{ (k, T(len, decay).toLong) }
  }._1 // "_1" projects to the first element of the tuple

def replicated[A,R](proc: A=>R)(argument:A, p:Double, k:Int) = {
  val lastPid = clock(p, dt())
  spawn[Long,A,R](pid => arg => (proc(arg), pid > lastPid-k),
    Set(lastPid), argument)
} // returns Map[Long,R] from replicate IDs to corresp. values
```

This function allows to replicate process `proc` every p seconds, keeping k replicates active at any time. It leverages a distributed `clock` function to provide a nearly-synchronised timer. The clock returns, each p seconds, an increasing number i that is also the PID of the ith replica. Notice that any device can locally determine when it can leave a process instance, with no risk for re-entrance (i.e., there is no need of *Prolonged Termination (Section 5.1.2)* and *Explicit Collective Termination (Section 5.1.3)*). Indeed, the status logic for participation to the process checks that the process has `pid` later than `lastPid-k`, namely is in the range $[\texttt{lastPid} - k + 1, \texttt{lastPid}]$ of the k currently valid PIDs.

Discussion. The implementation schema can be leveraged to implement the replicated gradient discussed previously and shown in Fig. 6.

```
val isSource: Boolean = ??? // some source definition logic
val REPLICATION_INTERVAL = 20 // twenty seconds
val NUM_REPLICATEs = 3 // three active replicas at a time
val replicatedGradientReplicas =
  replicated[Boolean,Double](distanceTo(_))
    (isSource, REPLICATION_INTERVAL, NUM_REPLICATES)
val replicatedGradient = replicatedGradientReplicas
  .minByOption(_._1) // takes the one with lowest PID
  .map(_._2) // projects the gradient value
  .getOrElse(Double.PositiveInfinity) // returns it
```

The generation of multiple replicas also requires to think about strategies to deal with their results. In the case of the replicated gradient, one simple idea is to use as gradient value the value of the oldest gradient replica, which is supposed to be the most stable. The oldest replica is considered for a duration given by the replication interval, which is used to regulate the reactivity of the system. The number of replicates, instead, is useful to guarantee enough time for replicas to stabilise before being actually considered.

5.1.5. Spatial replication

Name. *Spatial Replication*

Aliases. *Distributed Concurrent Processes*

Intent. The intent is to support the instantiation of multiple collective process instances across different portions of space or, similarly, across different sets of devices.

Context and example(s). Certain kinds of collective activities have to be executed concurrently in different spatial areas. Consider a smart city: the city may host, at different locations, multiple processes for surveillance, waste collection, monitoring, and management activities. Similarly, different process instances may be used to denote different areas or regions, each one hosting the same or different kinds of activities. One notable example is given by the Self-Organising Coordination Regions (SCR) pattern [68], which provides a dynamic way to control the level of decentralisation in systems of systems. Well, the regions of SCR can be denoted by different collective process instances. In other words, Spatial Replication is a pattern that can be used to implement SCR.

Description. The basic idea of the pattern is to consider multiple *regions* of devices (denoting spatial regions) and to run a different collective process instance in each region. Notice that a single region may be represented by a Boolean predicate: the devices for which the predicate yields `true` define the domain of the region (e.g., all the nodes located north of a geoposition, or where the sensed temperature is higher than a threshold). Thus, multiple regions may be denoted by multiple Boolean predicates. The regions may be

- contiguous or discontiguous;
- overlapping or non-overlapping (also: non-exclusive or exclusive);
- of different *shapes*.

The regions may be pre-defined or may be created and adjusted dynamically. In the latter case, one idea is to select a sparse set of devices (i.e., locations), and let the process instances spread from there.

Implementation. Consider:

```
// Location-based spatial replication
val locations: Set[GeoCoordinate] = ???
val closest: Option[GeoCoordiante] =
  closestLocation(locations,myLocation())
spawn(processLogic(_)(_), closest.toSet, Args())
```

The set of locations may be provided by a centralised management entity or through a gossip process. Each devices detects the closest location, and runs a copy of the process using the chosen location as PID. Therefore, each node joining the process for the same location will be able to interact with neighbour members.

As a variant, consider:

```
// Centre-based spatial replication
val centre: Boolean = ???
spawn(
  key => args => {
    val distanceToCentre = distanceTo(key==mid())
    // ...
    (outputLogic(),
      if(distanceToCentre <= threshold) true else false)
  },
  if(centre) Set(mid()) else Set(), Args())
```

Predicate `centre` is computed to hold `true` in correspondence of a subset of devices working as initiators of the regions. For instance, it might be the result of a leader election process [29,77]. The process instance of each region has as PID the identifier of its centre. The condition for being part of a process/region, then, might be that the `distanceToCentre` is less or equal than some `threshold` value, which may be used to control the granularity of spatial replication.

Discussion. This pattern can be based on static variants of the *Space-attached Process (Section 5.3.2)* and *Node-attached Process (Section 5.3.1)* mobility patterns. The pattern is often used to solve *coverage* problems or to exploit a *divide-et-impera* principle for solving situated tasks (e.g., in a smart city). In particular, the pattern is often combined with the *Leader-based Decision-Making (Section 5.2.1)* pattern, by which there is one or more leaders that partially centralise the collective decisions in each area. Concrete implementations of the pattern should generally establish the *granularity* of the spatial replication (i.e. how many replicas are there and what is their size). Advanced applications of the pattern may promote the self-adaptation of the sizes of the regions, as a way to optimise task allocation and to control the level of decentralisation in a cyber–physical ecosystem [68].

5.2. Behavioural patterns

Once a collective activity has been started, the group of participating entities have to work in a coordinated fashion: this may require making *group-wide decisions*. This is the problem of *collective decision-making* [78,79]. Of course, since multiple algorithms may exist for collective decision making, the patterns in this category are meant to generally support the *structuring* of large-scale group decision-making within collective processes; in other words, specific algorithms may be adopted in implementations of the patterns.

5.2.1. Leader-based decision-making

Name. *Leader-based Decision-Making*

Aliases. *Leader-regulated Areas*

Intent. The intent is to coordinate the activity of multiple distributed candidate decision-makers while avoiding conflict.

Context and example(s). A swarm of robots may define *teams* for handling different kinds of missions (e.g., rescue, exploration, or surveillance missions). Each team can be modelled as a collective process and, within each team, decisions may need to be taken about the tactics to adopt (e.g., among alternative courses of actions, such as continuing to explore the environment vs. going back to recharge). Each team may elect one leader to support decision-making within the team: then, the leader may make decisions using its local information only, by a democratic account of team members' opinions (cf. quorum sensing), or by using local reasoning against data provided by members. Similar needs may be found in a rather wide set of domains involving collectives: e.g., operations in a smart city, operations in industrial plants or warehouses, edge-cloud infrastructures, etc.

Description. The pattern works by solving the problem of distributed consensus through the centralisation of the decision-making activity at one device called the *leader* of a group of devices participating in a collective process instance. In other words, the patterns reduces the problem of decision-making to the problem of *adaptive leader election* [29,77]. The elected leader receives feedback information from the members of the collective process, processes the available information, takes a decision, and propagates the decision to the other members (which are instructed to follow the decision). Consider Fig. 7 for a graphical representation of the idea.

Any different process instance may influence the leader election process by the nature of the task it solves as well as its construction parameters. For instance, a collective process spanning large groups of devices or, similarly, large spatial regions, may

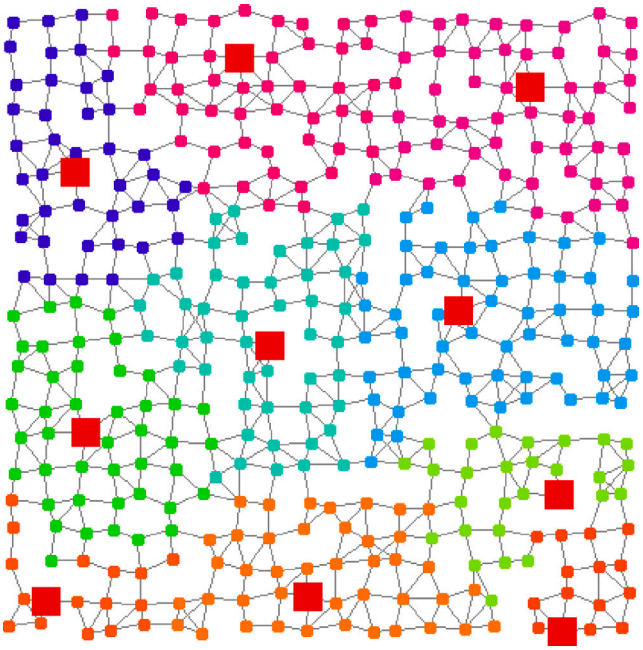


Fig. 7. In this simulation, the nodes elect a set of leaders (red squares) to uniformly cover the space occupied by the network itself. Each node participates to the collective process identified by the closest leader. Within each collective process, the corresponding leader propagates its decision (denoted by a different colour) to its followers. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

go for a *multi-leader election* in order to regulate the level of partitioning/granularity of decisions. Other times, there is a natural selection of a collective process leader, e.g., the one that generated the processes; in that case, the leader election activity itself it carried out outside the spawning of processes and, indeed, leveraged to regulate it.

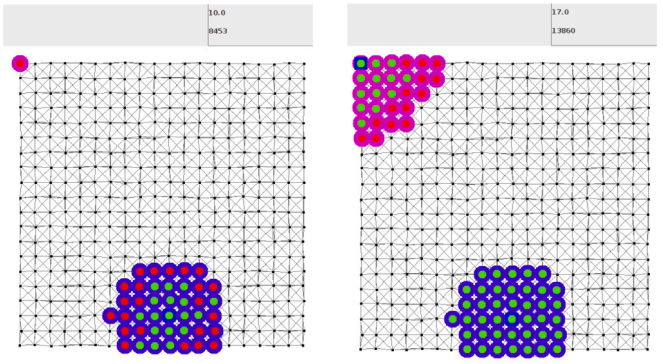
Implementation. A simple schema, where the leader election is performed outside the processes, and actually drives the generation of process instances, is as follows:

```
trait Decision
def decisionLogic(/*...*/): Decision
case class Pid(leaderId: ID)

val isLeader = leaderElection() // cf. [29]
val pids: Set[Pid] = if(isLeader) Set(mid()) else Set.empty

spawn(pid => args => {
  val pLeader = pid.leaderId == mid() && isLeader
  val localDecision: Decision = decisionLogic(/*...*/)
  val decision = broadcast(pLeader, localDecision)
  /* ... */
  // the output/status logic may depend on the decision
  (outputLogic(decision), statusLogic(decision))
}, pids, Args())
```

The election of leaders is performed by function `leaderElection()`, which returns a Boolean indicating whether the running device is a leader or not; such a field-calculus-based self-stabilising implementation of leader election is discussed in [29]. Within each process instance, `pLeader` carries out the Boolean indication of whether the running device (of device identifier `mid()`) is a leader or not *within the running process instance*. Notice how the lambda of the process logic closes over `isLeader` to account for removal of leadership, possibly to implement the process shutdown. Then, each device can take a `localDecision` by leveraging any local information. The ultimate decision that a device will make will depend on a multi-hop diffusion of the leader decision: the `broadcast` function uses a gradient from `pLeader` to propagate the value of `localDecision` at the source everywhere in the network.

An alternative schema is for the variant where leader election is performed internally to each instance:

```
case class Pid(electionControlParameter: Double)

spawn(pid => args => {
  val leader: ID = leaderElection(pid.electionControlParameter)
  val isLeader = leader == mid()
```



(a) One process instance (the blue-circled one at the bottom) is expanding when another one (the magenta one) springs out.

(b) The first process instance stabilises (cf. the green inner dots), whereas the second one is expanding (cf. the red inner dots denoting the evolving border).

(c) Both instances expand to gather more resources.

(d) Once done, both instances terminate (one completely, the other in the process).

Fig. 8. State-based collective behaviour. Notation: the outer circle denotes different process instances; the inner circle denotes the current state of the behaviour (red for "expansion state", green for "monitoring state"). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

```
  // decision propagation is the same as before
}, pids(), Args())
```

In this case, the PID may carry a control parameter affecting the `leaderElection()` computation (e.g., the mean size of the spatial area regulated by each leader).

Discussion. The leader of each process may regulate the decision-making of the activity but also the decision-making at the meta-level of the collective process, e.g., by determining its expansion, shrinking, or termination. This pattern is often combined with the *Spatial Replication* *(Section* 5.1.5*)* pattern, where each spatial replica has an associated leader that takes decisions within the corresponding region. If the regions overlap, then a criterion has to be defined so that the devices participating in multiple regions can determine what leader has to be followed. This is a very fundamental pattern, often reused by other patterns as well, such as the *State-based Collective Behaviour* *(Section* 5.2.2*)* pattern.

5.2.2. State-based collective behaviour

Name. *State-based Collective Behaviour*

Aliases. *Globally Synchronised Activity*

Intent. The intent is to describe an articulated collective behaviour and its *coordinated* evolution over time, e.g., based on the occurrence of local or global events.

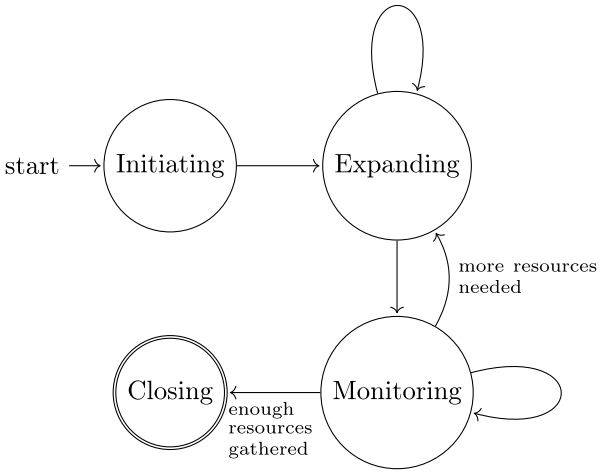


Fig. 9. State machine capturing the behaviour of a collective process that expands (i.e., acquires more devices) in discrete steps, until enough resources have been gathered.

Context and example(s). This pattern has been adopted in the literature in a few applications. One use is in the implementation of the *spatiotemporally situated tuples* model [15]: a fully decentralised spatial coordination model where data items (tuples) are given a location (or an area) in space, supporting interaction among mobile agents. In this case, each situated tuple is mapped to a collective process instance; then, the management of the operation for tuple removal requires all the agents associated with that tuple and involved in offering the coordination service to behave in synchronised phases (i.e., waiting for a removal operation, selecting a unique remover based on timeouts, offering the tuple, receipt of the acknowledgment from the remover, and shutdown). Another instance is in *peer-to-peer service discovery/access* [7]: agents can act as consumers or providers and enter in different phases of operation: *discovering* (looking for service offers), *offering* (providing matching service offers), *serving* (providing a chosen service).

Description. The pattern leverages the behavioural model of finite state machines: i.e., it works by specifying a set of collective behaviour components (associated to corresponding *states*) and the logic by which the collective behaviour switches between the behaviour components (states) over time or, generally, according to *events*. The collective behaviour change has to be coordinated, so that each collective behaviour component is played consistently; this is achieved by ensuring that the collective moves to the next state as a whole, namely that *all (or the majority of) the devices in a group eventually converge to the shifted behaviour*. The problem of determining the next state to switch on is essentially a problem of distributed consensus, or collective decision-making, and may be addressed through *Leader-based Decision-Making (Section 5.2.1)*.

Implementation. The implementation can benefit from a `workflow` function that (i) is parametrised on the state transition logic `f`, (ii) handles propagation of the current state (cf. the `broadcastOn` which spreads `currState` along a `gradient` from the leader), and (iii) ensures, by `align`ing on the state, that only the devices in the same state can interact at the level of function `f`.

```
def workflow[S, E](gradient: Double, initState: S, initResult: E)
                  (f: (S,E) => (S,E)): (S,E) = {
  rep[(S, E)]((initState, initResult)) { case (currState, r) => {
    val state = broadcastOn(gradient, currState)
    align(state) { s => f(s, r) }
  } }
}
```

Then, consider a sample scenario (cf. Fig. 8) where a leader has to collect a certain amount of resources from nearby devices. We can structure this logic by alternating two states: an `ExpandingState` where the frontier of devices gets expanded, and a `MonitoringState` where data is analysed for the current gathering. The behaviour can be described by a state machine as in Fig. 9.

```
spawn[Pid,Args,Return](pid => args => {
  val leader = mid() == pid.leaderId
  val g = distanceTo(leader)
  val (state, n) = workflow[State,Int](g, Initiating, 1) {
    case (Initiating, n) => (Expanding(n), n)
    case s @ (ExpandingState(k), r) => {
      val maxExtension = C[Double, Double](g, Math.max, g, g)
      if(leader && maxExtension >= (k - 1) * pid.distance) {
        (Monitoring(k), k)
      } else { (s, k) }
    }
```

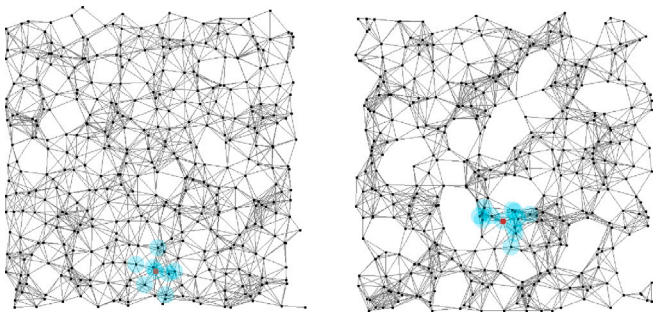


Fig. 10. A process (denoted by coloured circles) attached to a moving device (denoted by the small red circle). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

```
    case (s @ MonitoringState(k), r) => {
      val countMembers = C[Double,Int](g, _ + _, 1, 0)
      val isStable = stableFor(countMembers, pid.numRoundsOfStability)
      if(leader && isStable) {
        if(countMembers >= pid.resourceTarget) {
          (Closing, k) // once enough resources are gathered, terminate
        } else {
          (Expanding(k + 1), k + 1)
        }
      } else { (s, k) }
    }
    case s => s // stay in current state
  }
  val withinProcess = state != Closing && g < (n + 1) * pid.distance
  val leaderNotReentering = // cf. Prolonged Termination (Section 5.1.2)
  (outputLogic(), withinProcess && leaderNotReentering)
}, if(anchor) pidSet else Set.empty, Args())
```

The code builds a gradient `g` from the leader, then calls `workflow` by passing a partial function mapping each state to the corresponding output and next state. When in `ExpandingState`, the leader shifts to `Monitoring` once the maximum gradient value exceeds a threshold. Notice that output k denotes the levels of expansion, and acts as a multiplying factor of the distance threshold (`pid.distance`). When in `MonitoringState`, the leader collects (by `C`) the count of members in the process, and once the result is stable, it either shifts to `ExpandingState(k+1)` or to `ClosingState` if the amount of gathered resources exceeds `pid.resourceTarget`.

Discussion. This pattern helps to implement a collective activity by structuring it as a state machine. The provided implementation is combined with the *Leader-based Decision-Making (Section 5.2.1)*, to simplify collective transition to states, and *Prolonged Termination (Section 5.1.2)*, to ensure that once the leader closes the process, the decision is final.

The example in Fig. 8 shows an interesting application of the pattern: a step-wise progressive expansion of the process boundary to gather an increasing amount of participants/resources. Such an expansion is alternated with a phase in which the boundary does not change: such a stable situation helps to collect all relevant information into a stable snapshot for decision-making. This idea may be used to progressively gather computational resources in decentralised volunteer computing settings [54], before task allocation and scheduling. The state-based dynamics may also be used to implement wave-like dynamics, as in the *Moving Process (Section 5.3.3)* pattern.

5.3. Mobility patterns

Devices and processes may move or be stationary. Mobility patterns cover ways to address moving devices and processes.

5.3.1. Node-attached process

Name. *Node-attached Process*

Aliases. *Follower Process*

Intent. The intent is to ensure that a collective process follows (i.e. remains attached to) a moving device.

Context and example(s). Sometimes, a collective process is intimately related to the device that started it. For instance, a device may start a process to gather information from its surroundings (e.g., looking for nearby devices offering computational resources, or friends for social activities). A way to query a large portion of the system is to let such a process grow in size, and to collect information by descending the gradient through hop-by-hop paths. One drawback of this solution is that the devices that act as relays for the information coming from the more distant ones are exposed to such information, possibly leading to bandwidth (if several

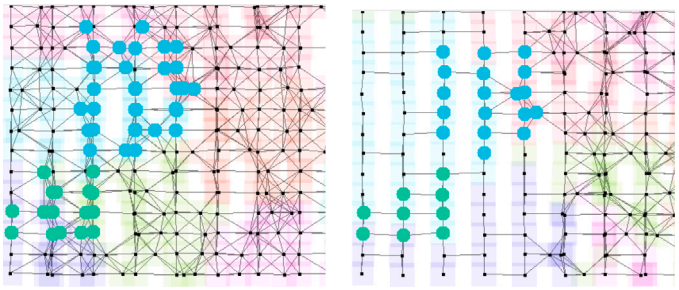


Fig. 11. A process attached to a spatial area. The semi-transparent shadows correspond to carrier processes (different colours for different carriers), whereas the solid circles denote space-attached processes (different colours for different processes). The two figures show two subsequent snapshots of a system where several devices move away from the areas with space-attached processes and accordingly leave the corresponding space-attached collective process computations (this is why in the right figure there are less devices denoting the green and cyan space-attached processes). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

such processes are active) as well as *privacy issues*. Asymmetric cryptography may be used for confidentiality, but it comes at high costs. Moreover, if the device is moving, then the scope of the process may need to adapt as well (as the notion of "surroundings" is changing).

Description. This pattern ensures that a collective process instance lives in the *surroundings* of a certain device (also called the *anchor*). If the anchor device moves, then the scope or domain of the process should change as well. The pattern does not prescribe a specific *shape* and *size* of the process—these are configurable properties. A typical shape is a circular bubble of a given (and possibly dynamic) radius. See Fig. 10 for a graphical example.

Implementation. The implementation is quite simple:

```
val anchor: Boolean = // some arbitrary condition
spawn[Pid,Args,Return](
  pid => args => {
    val source = mid()==pid && anchor
    val closeToAnchor = distanceTo(source) < SIZE
    (outputLogic(), closeToAnchor)
  },
  // just use the device ID as PID
  if(anchor) Set(Pid(mid())) else Set.empty, Args())
```

The simplicity is due to the self-healing nature of the gradient algorithm, `distanceTo`: if the anchor moves away from a region, then the gradient values computed by the devices in the original region would increase, eventually exceeding the threshold `SIZE`. Notice how the source is the device whose ID matches the PID *and* that it is still an anchor. Finally, the predicate `closeToAnchor` is also used as the spawn status, to determine if the device is part of the process instance or not.

Discussion. Despite its simplicity, the pattern is useful for addressing different needs in IoT environments. One natural use is to support *context awareness* [80]: any device that needs to be context-aware may use an attached process to continuously process information in its surroundings. Another use is to leave marks in the explored environment (cf. stigmergy [81,82]). An additional use is for *searching* resources or information, *by combining device mobility and the "process mobility"* given by multi-hop information gathering; device mobility may be required in scenarios where a network is sparse or largely disconnected (cf. *message ferrying* [83]).

5.3.2. Space-attached process

Name. *Space-attached Process*

Aliases. *Fixed-location Process*

Intent. The intent is to associate a collective process to a fixed spatial area, so that the devices that happen to be located there can handle a location-specific situation.

Context and example(s). In many applications, especially in the realm of *spatial computing* [84], the activities to be carried out strictly depend on the portion of the environment at hand. For instance, a critical area has to be surveilled, keeping track of accesses (cf. k-coverage [9]); a building has to be monitored for its internal climatic conditions [85]; or, an area hosting a large crowd of people has to provide crowd-aware navigation services, for safety [70]. Also, in programming models leveraging stigmergic coordination [81,82], data items may be associated to spatial locations and areas [15], to support, e.g., augmented-reality applications [86].

Description. The idea of this pattern is to establish a correspondence or *mapping* between *spatial areas* and collective processes, such that the devices that happen to be located in an area participate to the corresponding processes. The crux of the problem lies in how a device gets to know the mapping and whether its current location matches any of the appointed spatial areas. Also, the mapping might be subject of change at runtime. Different assumptions or constraints may affect the pattern solution: whether the devices known their current location (e.g., since they are equipped with GPS sensors) or not; or the kind of geometry that applies to the space (cf. the notion of vicinity inside buildings). Assuming that devices have a way to determine whether they are located in a given spatial area, the solution of the problem lies in organising a process providing the devices the mapping information. Such a collective process should have a spatial extension larger than the spatial areas provided in the mapping, in order to avoid the risk of losing track of the mapping information due to devices leaving those areas. The spatial area whereby mapping information is gossipped is also called the *carrier area*, and the devices contributing to holding the mapping information are called *relays*. Another aspect to be decided is the *issuer*, namely *who* is responsible of contributing to the mapping, i.e., of deciding what processes should be executed in what spatial areas: it could be anyone (cf. peer-to-peer systems), a set of designated devices (e.g., chosen by leader election), or a centralised manager. Furthermore, the device carrying the mapping information should be as stable as possible, to reduce the risk of information loss. As a graphical example, consider the simulation shown in Fig. 11: here, a number of nodes is instructed to migrate from the left to the right of the arena; however, only the nodes that remain fixed continue to run the process.

Implementation. An implementation schema for the pattern is as follows:

```
case class Pid(issuer: ID, area: AreaDescription) {
  def within(p: Position): Boolean = // matching logic
}
def issuing(): Set[Pid] = // generation of space-attached processes
val leader = // some leader election algorithm, e.g., see [29]
// Partition the space by the leader into carrier areas
val carrier = broadcast(leader, mid())
var pids: Set[Pid] = Set.empty // NOTE: mutable variable!
// Spawning of carrier processes
val carriers = spawn[CarrierPid,Unit,Unit](pid => args => {
  pids = pids ++ gossip[Set[Pid]](issuing(), _++_)
  ((), leader || pid.leader == carrier)
}, if(leader) Set(CarrierPid(mid())) else Set.empty, {})
// Spawning of space-attached processes
val maps = spawn[Pid,Unit,Unit](pid => args => {
  (outputLogic(), pid.within(currentPosition()))
}, pids, Args())
```

The first `spawn` is used to generate carrier processes, so that the `gossip` is limited to a single carrier area, and accumulate in each device the `pids` to be run (notice that it is a closure, closing over the `pids` variable). The second `spawn` is used to generate space-attached processes, based on the `pids` collected from the gossip; a device participates to a process instance of a given `pid` only if its `currentPosition` lies `within` the `pid`'s `area`. Notice that only issuers will yield a non-empty set when calling `issuing`.

Discussion. One problem lies in how to *specify* the spatial regions in which a process instance should reside. One straightforward possibility is by using *spatial coordinates* (e.g., a set of latitude and longitude points). The other problem, as mentioned, lies in how the information about the spatially-attached processes gets spread in the system. In the proposed implementation, the *Spatial Replication (Section 5.1.5)* pattern is used to handle the propagation of such information. Once any device has access to the map of spatially-attached processes, the next problem is understanding whether it is *part* of some of them. The general assumption is that any device has a local sensor enabling to answer it (e.g., a GPS sensor). This requirement may sometimes be relaxed by leveraging neighbour information. Finally, notice that when the target spatial location is relative to the spatial location of another entity (e.g., within 50 metres from an appointed building or plaza), *Node-attached Process (Section 5.3.1)* may represent an alternative to this pattern—assuming there is a device, part of the aggregate computing system and attached to the target spatially situated entity.

5.3.3. Moving process

Name. *Moving Process*

Aliases. Specialisations include *Wave* [7] and *Channel* (a point-to-point stretched process).

Intent. The intent is to carry data and/or run an activity across a network of devices while keeping it scoped to a subset of devices at a time, e.g., for efficiency or coordination purposes.

Context and example(s). The idea is similar to those of *mobile agents*, migrating from nodes to nodes in order to perform tasks, but with the difference that what migrates is not a single agent but a collective process that, at any time, might involve a collaboration among the nodes currently being part of its domain. Consider a messaging application on a peer-to-peer network: sending a message from a source to a destination could be performed by broadcasting data across a *channel* [6], i.e., a hop-by-hop path of devices connecting the two endpoints. Such a channel can be implemented involving all the nodes of the network by computing two gradients (one from the source and one from the destination), one network-wide broadcast of the source-to-destination distance, and then leveraging the triangular inequality property (see e.g. [87]). This would be very inefficient, since one network-wide channel per source–destination pair needs to be computed. A more efficient solution, leveraging a central node and a gradient from it,

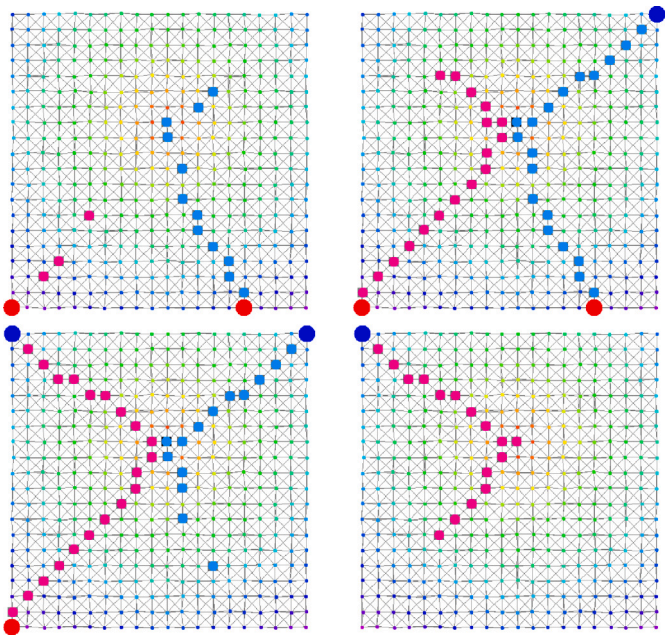


Fig. 12. Messaging on a peer-to-peer network. Notation: small coloured dots are devices, where the colour denotes the gradient from the central node; large red circles denote sources of messages; large blue circles denote targets of messages; medium-size bullets denote process domains, with different colours for different process instances (messages). The dynamics is as follows: (Top-left) two message delivery processes are active; (Top-right) the blue process has reached its target; (Bottom-left) the magenta process has reached its target, and the blue process is terminating from the source that received an acknowledgment of the delivery; (Bottom-right) the magenta process is also closing. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

would require building "narrow" channels along the source-to-centre and centre-to-destination paths—see Fig. 12. Still, if source and destination are far away, several hops and hence devices need to be involved, leading to a relevant consumption of resources associated to the long path of relays. An even more efficient solution could be based on a *wave-like* propagation of the message [7], being carried by a limited number of devices at any time.

Description. In this pattern, a process instance *moves* across space (or, equivalently, across a domain of connected devices). In particular, there are two main versions of the pattern:

1. the process moves by *changing its shape* (e.g., growing radially, elongating or stretching towards some direction, or shrinking);
2. the process moves by (approximately) *keeping its shape*.

Though the change in the support domain of devices is a normal aspect of a process instance, in a moving process there is usually a *direction* or *target* of the movement.

Implementation. This pattern can be implemented in multiple ways, mostly by devising proper algorithms regulating the way in which the devices compute the *status* (internal or external) in the process. Therefore, an implementation schema is as follows:

```
spawn(pid => args => {
  // ...
  val status = movementLogic(pid, args)
  (outputLogic(), status)
}, pids(), Args())
```

where an application-specific `movementLogic` is responsible of regulating the process shape.

Discussion. A moving process can be also realised by attaching a process to a moving device—cf. *Node-attached Process (Section 5.3.1)*. However, this Moving Process pattern also deals with moving collective computations within networks of *non-mobile devices*. How is a moving process different than a mere process expansion? The key distinguishing property of a moving process is that the expansion is *directional*. In more advanced versions of the pattern, process participants are *transient*, quitting the process instance as soon as they have moved it to neighbours ahead towards the target [7].

6. Case study: Spatiotemporal tuples

In this section, we describe a case study where several of the proposed patterns can find application. Indeed, our system is a pervasive coordination model in mobile settings where communication and interaction can be effectively mediated by collective

processes. As a consequence, the aspects captured by the taxonomy classes (lifecycle management, behaviour coordination, and mobility) are to be properly addressed.

The *spatiotemporal tuples* model [15], is a decentralised variant of the *spatial tuples* model [86], where tuples are issued at a spatial location, may have a spatial extension, and can be queried or removed by actions which are spatially scoped. These models support mobile augmented-reality applications, and support space-aware and space-based coordination of agents in pervasive computing environments. One example is situated knowledge sharing: in a rescue scenario, a rescue control room administrator may issue a warning to all the rescuer agents situated in a region, communicating the presence of injured people in another region.

In the original proposal [86], the implementation is centralised, whereas in [15], an aggregate computing implementation is provided that supports decentralised management of situated tuples. It turns out that such an implementation leverages multiple patterns of decentralised collective processes.

First of all, *all tuple operations map to distinct collective process instances*: this is because tuple operations must span different spatio-temporal regions, and hence must be carried out by collectives of situated devices. In a given application, tuple operations are issued as requested by the user (cf. the *Event-based Triggering (Section 5.1.1)*).

Then, different collective processes must model different tuple-based operations. In tuple-based coordination, three main operations are available: `out` (emission of a tuple at some location), `rd` (reading a tuple matching a pattern), `in` (removing a tuple matching a pattern). In spatial tuples, `out`s can be issued at a given spatial location (cf. the *Space-attached Process (Section 5.3.2)* pattern), can be associated to a device (cf. the *Node-attached Process (Section 5.3.1)* pattern), or can be given a complex spatiotemporal dynamics (cf. the *Moving Process (Section 5.3.3)* pattern).

Also, the implementation of `out`s and `in`s can be effectively organised following the *State-based Collective Behaviour (Section 5.2.2)* pattern: this is because a complex protocol should be followed to ensure that removal is performed by a single device, and that concurrent `in`s do not conflict (see Fig. 14). Specifically, an `out` operation should transit from an `Available` state (where it accepts removal requests), to a `Serving` state (where it has chosen a target device for the removal), a finally to a `Done` state (where it has received the acknowledgment by the remover and can therefore quit the process). Dually, the `in` operation would transit from a `Waiting` state (where a matching tuple is looked for), to a `Removing` state (where a chosen tuple is being removed and an acknowledgment is sent), to a `Done` state. To simplify the state-based behaviour implementation, there is a single tuple owner per `out` process, enabling the use of the *Leader-based Decision-Making (Section 5.2.1)* pattern, which removes the problem of consensus (e.g., in the selection of the "winner" `in` request). When closing the processes, the owner can issue a termination signal (cf. the *Explicit Collective Termination (Section 5.1.3)* pattern) and may adopt the *Prolonged Termination (Section 5.1.2)* pattern to avoid re-issuing the same tuple.

Finally, the *Spatial Replication (Section 5.1.5)* and *Time replication (Section 5.1.4)* patterns may also be leveraged if an application requires strategies for *replicating tuples* around the system. This could be especially useful when "evolving tuples" or "context-aware tuples" are considered (cf. TOTA [82]), so that different timing and spatial aspects could affect their content.

7. Discussion

In this section, we provide a broad discussion of the pattern catalogue presented in Section 5.

7.1. Relationships among the patterns of the catalogue

Pattern catalogues often promote *systems of patterns*, that may be synergistically used to address design in specific contexts. The provided patterns of collective processes aim to support the engineering of collective adaptive and self-organising behaviours.

Fig. 13 shows how the provided patterns are related. The graph is synthesised based on the references found in the description subsections of each pattern.

7.2. Applicability

The discussed patterns can be considered when designing *self-organising distributed systems* [88] and *cyber–physical collectives* [89] in general. Though the implementation schemas are given based on the aggregate computing model (cf. Section 3.1), namely assuming event structures of sense–compute–interact events [7], it does not mean that other similar models and programming languages could not benefit from them (cf. the Buzz swarm programming language [12]).

7.3. Macro-programming perspective

We argue that the proposed catalogue could be a starting point for *macro-programming patterns*, namely design patterns for promoting desired macroscopic properties and dynamics out of collectives of objects and agents. Though in the last decades, several macro-programming approaches have been proposed (cf. see [20] for recent reviews), very few attempts have been made to characterise reusable macro-level patterns.

One key message of this paper is that the *macroscopic perspective* could *complement* the traditional microscopic perspective, where the target of the engineering is the individual device. A peculiar aspect of macro-programming is, indeed, the use of *macro-programming abstractions* [20], of which, e.g., *spatial abstractions* [22] form a subset. The proposed patterns define a *vocabulary* of design notions where terms are at the macro or collective level: i.e., they talk about what entire groups of agents should do, rather than what a single individual must perform. Accordingly, these terms promote reasoning and designing at a *global* (or system-wide) level.

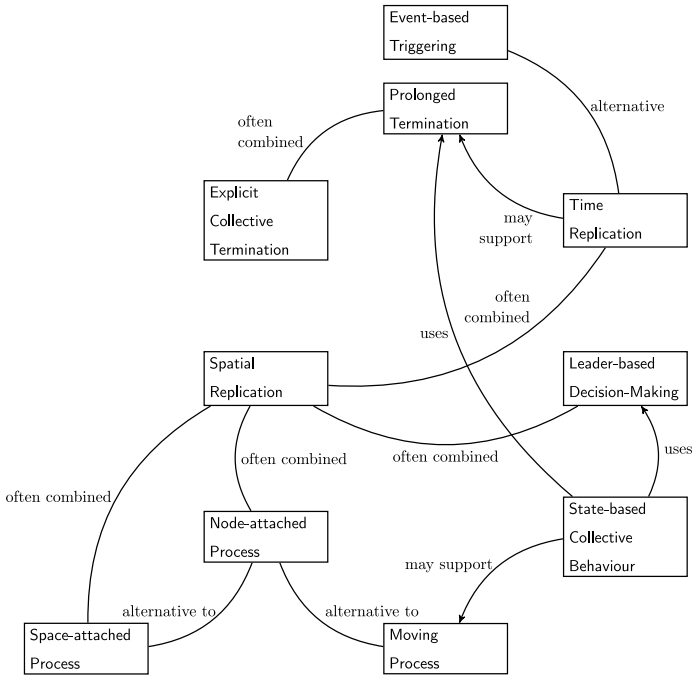


Fig. 13. This diagram shows how the patterns are related to one another.

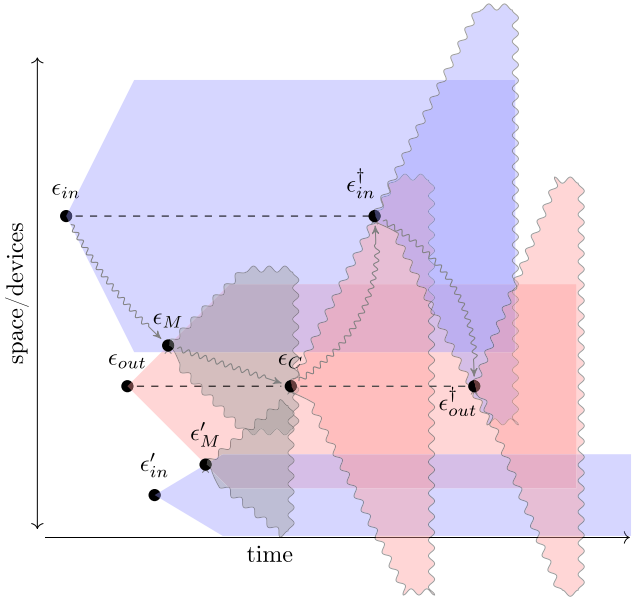


Fig. 14. (Picture from [15]) Spatiotemporal tuples: a removal operation and the interaction between one `out` process instance (issued at event ϵ_{out}, and shown in red) and two `in` processes instances (issued at ϵ_{in} and ϵ'_{in}, and shown in blue). Notice how only the former `in` process terminates, whereas the one issued at ϵ'_{in} continues to live, looking for another tuple to remove. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

7.4. Limitations and threats to validity

The work focusses on decentralised collective process patterns based on the aggregate process programming model [6] (cf. Section 3.3). As a consequence, the scope is somewhat limited in the target systems and applications covered by the patterns, which are mainly those characterised by self-organisation of large-scale homogeneous systems (rather than, e.g., small-scale functional composites made of highly heterogeneous devices). Also, the proposed catalogue does not consider patterns arising from the

interaction among multiple collective processes; since research is limited on the topic, such kinds of patterns will be investigated in the future.

Internal validity is about the extent to which the study design may undermine the conclusions [90], and is threatened e.g. by systematic errors (bias), missing factors, and simplifications to the model. Regarding internal validity, the arguments about the macro-programming stance (cf. Section 7.3) are threatened by the lack of formal definitions clarifying in what sense a piece of code is at the macro level. However, since the programming model leverages *fields* as collective data structures (cf. Section 3.2), and aggregate processes are effectively first-class entities, this view is arguably supported. As a contingency measure for the identification and classification of patterns, two experts (one in swarm robotics and the other in multi-agent planning), both familiar with aggregate processes, were involved in a discussion with the author, which ultimately led to keeping the proposed pattern catalogue and taxonomy unchanged.

External validity refers to the extent to which the results can be generalised [90]. Regarding external validity, the extent to which the proposed patterns can be transferred to different collective computing and programming models may be unclear. This issue is exacerbated by the fact that macro-programming models tend to be quite ad-hoc [20]. Indeed, the proposed patterns are quite specific to the aggregate process programming model, and certain patterns (like *Explicit Collective Termination (Section 5.1.3)* and *Prolonged Termination (Section 5.1.2)*) may also be seen as induced by inherent idiosyncrasies of the `spawn` construct. However, the *universality* of field calculi [62] may be seen as a mitigating factor for this threat. Variants of field calculi based on the same execution model [87] or variants of the round-based execution model (e.g., reactive declinations [91]) are being explored, but they retain substantial similarity. In other words, the adopted computational model and language appear quite *canonical* for programming decentralised collective processes.

8. Conclusion

In this work, we have proposed a catalogue of patterns for decentralised collective processes. The proposed items can be considered as *design patterns* for collective behaviour as well as *coding patterns* based on aggregate computing and the aggregate process abstraction [6,7]. Notice that these patterns are not invented but rather *discovered* after several works adopting macro-programming, aggregate computing, and aggregate processes [6,7,31,65,66]. Accordingly, the catalogue can be regarded as a contribution to research on language-based software engineering [92] and especially on macro-programming [20,21,63].

This work promotes further research contributions. For instance, the proposed pattern could be integrated into the standard libraries of aggregate programming languages like ScaFi [32], FRASP [91], Protelis [93], FCPP [94]. More patterns can also be discovered. One class of patterns that has not been considered in this paper includes *control patterns* aimed at the management of the interaction among different collective processes. In this perspective, drawing connections with research work on business process modelling [95] may be valuable. A more innovative direction may lie in the use of patterns to help the synthesis of programs for controlling collective behaviour [96–98]. Finally, it would be interesting to study and evaluate the applicability of the proposed patterns, e.g., following evidence-based frameworks like [99,100].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared my code at a public permanent archive: https://zenodo.org/records/11210637.

References

[1] L. Atzori, A. Iera, G. Morabito, The Internet of Things: A survey, Comput. Netw. 54 (15) (2010) 2787–2805, http://dx.doi.org/10.1016/j.comnet.2010.05.010.

[2] G.E. Mobus, Principles of systems science, in: Systems Science: Theory, Analysis, Modeling, and Design, Springer International Publishing, 2022, pp. 41–87, http://dx.doi.org/10.1007/978-3-030-93482-8_2.

[3] C.B. Nielsen, P.G. Larsen, J.S. Fitzgerald, J. Woodcock, J. Peleska, Systems of systems engineering: Basic concepts, model-based techniques, and research directions, ACM Comput. Surv. 48 (2) (2015) 18:1–18:41, http://dx.doi.org/10.1145/2794381.

[4] O. Zedadra, A. Guerrieri, N. Jouandeau, G. Spezzano, H. Seridi, G. Fortino, Swarm intelligence-based algorithms within IoT-based systems: A review, J. Parallel Distrib. Comput. 122 (2018) 173–187, http://dx.doi.org/10.1016/j.jpdc.2018.08.007.

[5] G.D. Abowd, Beyond Weiser: From ubiquitous to collective computing, Computer 49 (1) (2016) 17–23, http://dx.doi.org/10.1109/MC.2016.22.

[6] R. Casadei, M. Viroli, G. Audrito, D. Pianini, F. Damiani, Engineering collective intelligence at the edge with aggregate processes, Eng. Appl. Artif. Intell. 97 (2021) 104081, http://dx.doi.org/10.1016/j.engappai.2020.104081.

[7] G. Audrito, R. Casadei, G. Torta, A general framework and decentralised algorithms for collective computational processes, Future Gener. Comput. Syst. 158 (2024) 11–27, http://dx.doi.org/10.1016/j.future.2024.04.020.

[8] M.T. Lazarescu, Design of a WSN platform for long-term environmental monitoring for IoT applications, IEEE J. Emerg. Sel. Top. Circuits Syst. 3 (1) (2013) 45–54, http://dx.doi.org/10.1109/JETCAS.2013.2243032.

[9] D. Pianini, F. Pettinari, R. Casadei, L. Esterle, A collective adaptive approach to decentralised k-coverage in multi-robot systems, ACM Trans. Auton. Adapt. Syst. 17 (2022) 4:1–4:39, http://dx.doi.org/10.1145/3547145.

[10] Y.A. Alrahman, R.D. Nicola, M. Loreti, Programming interactions in collective adaptive systems by relying on attribute-based communication, Sci. Comput. Program. 192 (2020) 102428, http://dx.doi.org/10.1016/j.scico.2020.102428.
[11] C. Razafimandimby, V. Loscrì, A.M. Vegni, A neural network and IoT based scheme for performance assessment in Internet of Robotic Things, in: First IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016, IEEE Computer Society, 2016, pp. 241–246, http://dx.doi.org/10.1109/IoTDI.2015.10.
[12] C. Pinciroli, G. Beltrame, Buzz: An extensible programming language for heterogeneous swarm robotics, in: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2016, IEEE, 2016, pp. 3794–3800, http://dx.doi.org/10.1109/IROS.2016.7759558.
[13] A. Shahraki, A. Taherkordi, Ø. Haugen, F. Eliassen, A survey and future directions on clustering: From WSNs to IoT and modern networking paradigms, IEEE Trans. Netw. Serv. Manag. 18 (2) (2021) 2242–2274, http://dx.doi.org/10.1109/TNSM.2020.3035315.
[14] G. Aguzzi, G. Audrito, R. Casadei, F. Damiani, G. Torta, M. Viroli, A field-based computing approach to sensing-driven clustering in robot swarms, Swarm Intell. 17 (1) (2023) 27–62, http://dx.doi.org/10.1007/s11721-022-00215-y.
[15] R. Casadei, M. Viroli, A. Ricci, G. Audrito, Tuple-based coordination in large-scale situated systems, in: Coordination Models and Languages - 23rd International Conference, COORDINATION 2021, Proceedings, in: LNCS, vol. 12717, Springer, 2021, pp. 149–167, http://dx.doi.org/10.1007/978-3-030-78142-2_10.
[16] B. Guo, Y. Liu, S. Liu, Z. Yu, X. Zhou, CrowdHMT: Crowd intelligence with the deep fusion of human, machine, and IoT, IEEE Internet Things J. 9 (24) (2022) 24822–24842, http://dx.doi.org/10.1109/JIOT.2022.3194726.
[17] O. Scekic, T. Schiavinotto, S. Videnov, M. Rovatsos, H.L. Truong, D. Miorandi, S. Dustdar, A programming model for hybrid collaborative adaptive systems, IEEE Trans. Emerg. Top. Comput. 8 (1) (2020) 6–19, http://dx.doi.org/10.1109/TETC.2017.2702578.
[18] M. Brambilla, E. Ferrante, M. Birattari, M. Dorigo, Swarm robotics: a review from the swarm engineering perspective, Swarm Intell. 7 (1) (2013) 1–41, http://dx.doi.org/10.1007/s11721-012-0075-2.
[19] R.D. Nicola, S. Jähnichen, M. Wirsing, Rigorous engineering of collective adaptive systems: special section, Int. J. Softw. Tools Technol. Transf. 22 (4) (2020) 389–397, http://dx.doi.org/10.1007/s10009-020-00565-0.
[20] R. Casadei, Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling, ACM Comput. Surv. (2023) http://dx.doi.org/10.1145/3579353.
[21] I.G.S. Júnior, T.S. Santana, R.F. Bulcão-Neto, B. Porter, The state of the art of macroprogramming in IoT: An update, J. Internet Serv. Appl. 13 (1) (2022) 54–65, http://dx.doi.org/10.5753/jisa.2022.2372.
[22] J. Beal, S. Dulman, K. Usbeck, M. Viroli, N. Correll, Organizing the aggregate: Languages for spatial computing, in: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, IGI Global, 2013, pp. 436–501, http://dx.doi.org/10.4018/978-1-4666-2092-6.ch016.
[23] J.L. Fernandez-Marquez, G.D. Serugendo, S. Montagna, M. Viroli, J.L. Arcos, Description and composition of bio-inspired design patterns: a complete overview, Nat. Comput. 12 (1) (2013) 43–67, http://dx.doi.org/10.1007/s11047-012-9324-y.
[24] Ö. Babaoglu, G. Canright, A. Deutsch, G. Di Caro, F. Ducatelle, L.M. Gambardella, N. Ganguly, M. Jelasity, R. Montemanni, A. Montresor, T. Urnes, Design patterns from biology for distributed computing, ACM Trans. Auton. Adapt. Syst. 1 (1) (2006) 26–66, http://dx.doi.org/10.1145/1152934.1152937.
[25] M. Viroli, G. Audrito, J. Beal, F. Damiani, D. Pianini, Engineering resilient collective adaptive systems by self-stabilisation, ACM Trans. Model. Comput. Simul. 28 (2) (2018) 16:1–16:28, http://dx.doi.org/10.1145/3177774.
[26] R. Nagpal, A catalog of biologically-inspired primitives for engineering self-organization, in: Engineering Self-Organising Systems, Nature-Inspired Approaches To Software Engineering [Revised and Extended Papers Presented at the Engineering Self-Organising Applications Workshop, ESOA 2003, in: LNCS, vol. 2977, Springer, 2003, pp. 53–62, http://dx.doi.org/10.1007/978-3-540-24701-2_4.
[27] T.D. Wolf, T. Holvoet, 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, IEEE Computer Society, 2007, pp. 295–298, http://dx.doi.org/10.1109/SASO.2007.16.
[28] G. Audrito, R. Casadei, F. Damiani, D. Pianini, M. Viroli, Optimal resilient distributed data collection in mobile edge environments, Comput. Electr. Eng. 96 (2021) 107580, http://dx.doi.org/10.1016/j.compeleceng.2021.107580.
[29] D. Pianini, R. Casadei, M. Viroli, Self-stabilising priority-based multi-leader election and network partitioning, in: IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2022, IEEE, 2022, pp. 81–90, http://dx.doi.org/10.1109/ACSOS55765.2022.00026.
[30] H. Oh, A.R. Shirazi, C. Sun, Y. Jin, Bio-inspired self-organising multi-robot pattern formation: A review, Robotics Auton. Syst. 91 (2017) 83–100, http://dx.doi.org/10.1016/j.robot.2016.12.006.
[31] L. Testa, G. Audrito, F. Damiani, G. Torta, Aggregate processes as distributed adaptive services for the industrial Internet of Things, Pervasive Mob. Comput. 85 (2022) 101658, http://dx.doi.org/10.1016/J.PMCJ.2022.101658.
[32] R. Casadei, M. Viroli, G. Aguzzi, D. Pianini, ScaFi: A Scala DSL and toolkit for aggregate programming, SoftwareX 20 (2022) 101248, http://dx.doi.org/10.1016/j.softx.2022.101248.
[33] Anonymous, System-wide IoT design and programming: Patterns for decentralised collective processes, 2024, URL https://anonymous.4open.science/r/experiment-2023-collective-process-patterns-84FF.
[34] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture, Volume 1: A System of Patterns, Wiley, 1996.
[35] C. Alexander, S. Ishikawa, M. Silverstein, A Pattern Language: Towns, Buildings, Construction, Oxford University Press, 1977.
[36] E. Gamma, R. Helm, R. Johnson, J.M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, first ed., Addison-Wesley Professional, 1994.
[37] D.C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects, Wiley, 2000.
[38] G. Hohpe, B. Woolf, Enterprise integration patterns, The Addison-Wesley Signature Series, Prentice Hall, 2004.
[39] V. Vernon, Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka, first ed., Addison-Wesley Professional, 2015.
[40] M. Casciaro, Node.js Design Patterns, second Edition, Community Experience Distilled, Packt, 2016.
[41] R. Hanmer, Patterns for Fault Tolerant Software, Wiley Publishing, 2007.
[42] M.J. Pont, Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers, ACM Press/Addison-Wesley Publishing Co., USA, 2001.
[43] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg, Internet of Things patterns, in: Proceedings of the 21st European Conference on Pattern Languages of Programs, EuroPLoP 2016, ACM, 2016, p. 5, http://dx.doi.org/10.1145/3011784.3011789.
[44] C. Krupitzer, T. Temizer, T. Prantl, C. Raibulet, An overview of design patterns for self-adaptive systems in the context of the Internet of Things, IEEE Access 8 (2020) 187384–187399, http://dx.doi.org/10.1109/ACCESS.2020.3031189.
[45] H. Washizaki, S. Ogata, A. Hazeyama, T. Okubo, E.B. Fernández, N. Yoshioka, Landscape of architecture and design patterns for IoT systems, IEEE Internet Things J. 7 (10) (2020) 10091–10101, http://dx.doi.org/10.1109/JIOT.2020.3003528.
[46] G. Bloom, B. Alsulami, E. Nwafor, I.C. Bertolotti, Design patterns for the industrial Internet of Things, in: 14th IEEE International Workshop on Factory Communication Systems, WFCS 2018, IEEE, 2018, pp. 1–10, http://dx.doi.org/10.1109/WFCS.2018.8402353.

[47] T.D. Wolf, T. Holvoet, Design patterns for decentralised coordination in self-organising emergent systems, in: Engineering Self-Organising Systems, 4th International Workshop, ESOA 2006, Hakodate, Japan, May 9, 2006, Revised and Invited Papers, 2006, pp. 28–49, http://dx.doi.org/10.1007/978-3-540-69868-5_3.
[48] D. Weyns, B.R. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, K.M. Göschka, On patterns for decentralized control in self-adaptive systems, in: Software Engineering for Self-Adaptive Systems II - International Seminar, Revised Selected and Invited Papers, in: LNCS, vol. 7475, Springer, 2010, pp. 76–107, http://dx.doi.org/10.1007/978-3-642-35813-5_4.
[49] V. Sithole, L. Marshall, Systematic methods for organising patterns for the Internet of Things: A preliminary exploration, Internet Things 11 (2020) 100268, http://dx.doi.org/10.1016/J.IOT.2020.100268.
[50] R. Casadei, G. Fortino, D. Pianini, A. Placuzzi, C. Savaglio, M. Viroli, A methodology and simulation-based toolchain for estimating deployment performance of smart collective services at the edge, IEEE Internet Things J. 9 (20) (2022) 20136–20148, http://dx.doi.org/10.1109/JIOT.2022.3172470.
[51] Z. Wood, A. Galton, A taxonomy of collective phenomena, Appl. Ontol. 4 (3–4) (2009) 267–292, http://dx.doi.org/10.3233/AO-2009-0071.
[52] M.D. Weiser, Some computer science issues in ubiquitous computing, ACM SIGMOBILE Mob. Comput. Commun. Rev. 3 (3) (1999) 12, http://dx.doi.org/10.1145/329124.329127.
[53] M. Nasir, K. Muhammad, J. Lloret, A.K. Sangaiah, M. Sajjad, Fog computing enabled cost-effective distributed summarization of surveillance videos for smart cities, J. Parallel Distrib. Comput. 126 (2019) 161–170, http://dx.doi.org/10.1016/j.jpdc.2018.11.004.
[54] C. Funai, C. Tapparello, H. Ba, B. Karaoglu, W.B. Heinzelman, Extending volunteer computing through mobile ad hoc networking, in: IEEE Global Communications Conference, GLOBECOM 2014, Austin, TX, USA, December 8-12, 2014, IEEE, 2014, pp. 32–38, http://dx.doi.org/10.1109/GLOCOM.2014.7036780.
[55] R. Casadei, M. Viroli, Coordinating computation at the edge: a decentralized, self-organizing, spatial approach, in: Fourth International Conference on Fog and Mobile Edge Computing, FMEC 2019, IEEE, 2019, pp. 60–67, http://dx.doi.org/10.1109/FMEC.2019.8795355.
[56] V. Karagiannis, S. Schulte, Distributed algorithms based on proximity for self-organizing fog computing systems, Pervasive Mob. Comput. 71 (2021) 101316, http://dx.doi.org/10.1016/j.pmcj.2020.101316.
[57] G. Audrito, R. Casadei, F. Damiani, V. Stolz, M. Viroli, Adaptive distributed monitors of spatial properties for cyber-physical systems, J. Syst. Softw. 175 (2021) 110908, http://dx.doi.org/10.1016/j.jss.2021.110908.
[58] J. Liu, H. Shen, H.S. Narman, W. Chung, Z. Lin, A survey of mobile crowdsensing techniques: A critical component for the Internet of Things, ACM Trans. Cyber Phys. Syst. 2 (3) (2018) 18:1–18:26, http://dx.doi.org/10.1145/3185504.
[59] K. Zhang, Z. Yang, T. Basar, Multi-agent reinforcement learning: A selective overview of theories and algorithms, 2019, CoRR abs/1911.10635, arXiv:1911.10635.
[60] L. Busoniu, R. Babuska, B.D. Schutter, A comprehensive survey of multiagent reinforcement learning, IEEE Trans. Syst. Man Cybern. Part C 38 (2) (2008) 156–172, http://dx.doi.org/10.1109/TSMCC.2007.913919.
[61] D.H. Wolpert, K. Tumer, An Introduction to Collective Intelligence, 1999, CoRR cs.LG/9908014.
[62] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, D. Pianini, From distributed coordination to field calculus and aggregate computing, J. Log. Algebraic Methods Program. 109 (2019) http://dx.doi.org/10.1016/j.jlamp.2019.100486.
[63] R. Newton, M. Welsh, Region streams: Functional macroprogramming for sensor networks, in: Workshop on Data Management for Sensor Networks, 2004, pp. 78–87, http://dx.doi.org/10.1145/1052199.1052213.
[64] R. Casadei, M. Viroli, G. Audrito, D. Pianini, F. Damiani, Aggregate processes in field calculus, in: Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held As Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Proceedings, in: LNCS, vol. 11533, Springer, 2019, pp. 200–217, http://dx.doi.org/10.1007/978-3-030-22397-7_12.
[65] G. Aguzzi, R. Casadei, D. Pianini, M. Viroli, Dynamic decentralization domains for the Internet of Things, IEEE Internet Comput. 26 (6) (2022) 16–23, http://dx.doi.org/10.1109/MIC.2022.3216753.
[66] G. Aguzzi, R. Casadei, M. Viroli, MacroSwarm: A field-based compositional framework for swarm programming, in: Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, Discotec 2023, Proceedings, in: LNCS, 13908, Springer, 2023, pp. 31–51, http://dx.doi.org/10.1007/978-3-031-35361-1_2.
[67] G. Aguzzi, R. Casadei, D. Pianini, M. Viroli, Dynamic decentralization domains for the Internet of Things, IEEE Internet Comput. (2022) 1–10, http://dx.doi.org/10.1109/mic.2022.3216753.
[68] D. Pianini, R. Casadei, M. Viroli, A. Natali, Partitioned integration and coordination via the self-organising coordination regions pattern, Future Gener. Comput. Syst. 114 (2021) 44–68, http://dx.doi.org/10.1016/j.future.2020.07.032.
[69] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, Workflow Patterns: The Definitive Guide, MIT Press, 2016, URL https://mitpress.mit.edu/books/workflow-patterns.
[70] R. Casadei, G. Fortino, D. Pianini, W. Russo, C. Savaglio, M. Viroli, A development approach for collective opportunistic edge-of-things services, Inform. Sci. 498 (2019) 154–169, http://dx.doi.org/10.1016/J.INS.2019.05.058.
[71] M.N. Durrani, J.A. Shamsi, Volunteer computing: requirements, challenges, and solutions, J. Netw. Comput. Appl. 39 (2014) 369–380, http://dx.doi.org/10.1016/j.jnca.2013.07.006.
[72] R. Casadei, A. Aldini, M. Viroli, Towards attack-resistant aggregate computing using trust mechanisms, Sci. Comput. Program. 167 (2018) 114–137, http://dx.doi.org/10.1016/j.scico.2018.07.006.
[73] G. Audrito, R. Casadei, G. Torta, On the dynamic evolution of distributed computational aggregates, in: IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion, ACSOS-C 2022, IEEE, 2022, pp. 37–42, http://dx.doi.org/10.1109/ACSOSC56246.2022.00024.
[74] D. Pianini, J. Beal, M. Viroli, Improving gossip dynamics through overlapping replicates, in: Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held As Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Proceedings, in: LNCS, vol. 9686, Springer, 2016, pp. 192–207, http://dx.doi.org/10.1007/978-3-319-39519-7_12.
[75] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565, http://dx.doi.org/10.1145/359545.359563.
[76] D. Pianini, R. Casadei, M. Viroli, S. Mariani, F. Zambonelli, Time-fluid field-based coordination through programmable distributed schedulers, Log. Methods Comput. Sci. 17 (4) (2021) http://dx.doi.org/10.46298/LMCS-17(4:13)2021.
[77] Y. Mo, G. Audrito, S. Dasgupta, J. Beal, Near-optimal knowledge-free resilient leader election, Automatica 146 (2022) 110583, http://dx.doi.org/10.1016/j.automatica.2022.110583.
[78] A. Reina, E. Ferrante, G. Valentini, Collective decision-making in living and artificial systems: editorial, Swarm Intell. 15 (1–2) (2021) 1–6, http://dx.doi.org/10.1007/S11721-021-00195-5.
[79] Q. Zhan, C. Fu, M. Xue, Distance-based large-scale group decision-making method with group influence, Int. J. Fuzzy Syst. 23 (2) (2021) 535–554, http://dx.doi.org/10.1007/s40815-020-00993-9.
[80] O. Yurur, C.H. Liu, Z. Sheng, V.C.M. Leung, W.A. Moreno, K.K. Leung, Context-awareness for mobile sensing: A survey and future directions, IEEE Commun. Surv. Tutorials 18 (1) (2016) 68–93, http://dx.doi.org/10.1109/COMST.2014.2381246.
[81] K. Hadeli, P. Valckenaers, M.J. Kollingbaum, H.V. Brussel, Multi-agent coordination and control using stigmergy, Comput. Ind. 53 (1) (2004) 75–96, http://dx.doi.org/10.1016/S0166-3615(03)00123-4.

[82] M. Mamei, F. Zambonelli, Programming pervasive and mobile computing applications: The TOTA approach, ACM Trans. Softw. Eng. Methodol. 18 (4) (2009) 1–56, http://dx.doi.org/10.1145/1538942.1538945.
[83] W. Zhao, M.H. Ammar, E.W. Zegura, A message ferrying approach for data delivery in sparse mobile ad hoc networks, in: Proceedings of the 5th ACM Interational Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc 2004, ACM, 2004, pp. 187–198, http://dx.doi.org/10.1145/989459.989483.
[84] M. Duckham, Decentralized Spatial Computing: Foundations of Geosensor Networks, Springer, 2013.
[85] W. Sung, S. Hsiao, Building an indoor air quality monitoring system based on the architecture of the Internet of Things, EURASIP J. Wirel. Commun. Netw. 2021 (1) (2021) 153, http://dx.doi.org/10.1186/s13638-021-02030-1.
[86] A. Ricci, M. Viroli, A. Omicini, S. Mariani, A. Croatti, D. Pianini, Spatial tuples: Augmenting reality with tuples, Expert Syst. J. Knowl. Eng. 35 (5) (2018) http://dx.doi.org/10.1111/exsy.12273.
[87] G. Audrito, R. Casadei, F. Damiani, M. Viroli, Computation against a neighbour: Addressing large-scale distribution and adaptivity with functional programming and scala, Log. Methods Comput. Sci. 19 (1) (2023) http://dx.doi.org/10.46298/lmcs-19(1:6)2023.
[88] C. Prehofer, C. Bettstetter, Self-organization in communication networks: principles and design paradigms, IEEE Commun. Mag. 43 (7) (2005) 78–85, http://dx.doi.org/10.1109/MCOM.2005.1470824.
[89] R. Casadei, L. Esterle, R. Gamble, P. Harvey, E.F. Wanner, Editorial: Understanding and engineering cyber-physical collectives, Front. Robotics AI 11 (2024) http://dx.doi.org/10.3389/frobt.2024.1407421.
[90] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer, 2012, http://dx.doi.org/10.1007/978-3-642-29044-2.
[91] R. Casadei, F.L.D. Angelis, G. Aguzzi, D. Pianini, M. Viroli, Self-organisation programming: A functional reactive macro approach, in: IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2023, IEEE, 2023, pp. 87–96, http://dx.doi.org/10.1109/ACSOS58161.2023.00026.
[92] G. Gupta, Language-based software engineering, Sci. Comput. Program. 97 (2015) 37–40, http://dx.doi.org/10.1016/j.scico.2014.02.010.
[93] D. Pianini, M. Viroli, J. Beal, Protelis: practical aggregate programming, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, ACM, 2015, pp. 1846–1853, http://dx.doi.org/10.1145/2695664.2695913.
[94] G. Audrito, FCPP: an efficient and extensible field calculus framework, in: International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS, IEEE, 2020, pp. 153–159, http://dx.doi.org/10.1109/ACSOS49614.2020.00037.
[95] D. Kim, Y.K. Chung, R-BPMN for abstract modeling of business process patterns, Bus. Process. Manag. J. 27 (5) (2021) 1445–1462, http://dx.doi.org/10.1108/BPMJ-08-2020-0371.
[96] S. Gulwani, O. Polozov, R. Singh, Program synthesis, Found. Trends Program. Lang. 4 (1–2) (2017) 1–119, http://dx.doi.org/10.1561/2500000010.
[97] G. Aguzzi, R. Casadei, M. Viroli, Towards reinforcement learning-based aggregate computing, in: 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, Proceedings, in: LNCS, vol. 13271, Springer, 2022, pp. 72–91, http://dx.doi.org/10.1007/978-3-031-08143-9_5.
[98] C.W. Kessler, Pattern-driven automatic program transformation and parallelization, in: 3rd Euromicro Workshop on Parallel and Distributed Processing, (PDP'95), IEEE Computer Society, 1995, pp. 76–83, http://dx.doi.org/10.1109/EMPDP.1995.389152.
[99] B.A. Kitchenham, D. Budgen, P. Brereton, Evidence-Based Software Engineering and Systematic Reviews, Chapman & Hall/CRC, 2015.
[100] M. Riaz, T. Breaux, L. Williams, How have we evaluated software pattern application? A systematic mapping study of research design practices, Inf. Softw. Technol. 65 (2015) 14–38, http://dx.doi.org/10.1016/j.infsof.2015.04.002.