Programming Distributed Collective Processes for Dynamic Ensembles and Collective Tasks

Programming Distributed Collective Processes for Dynamic Ensembles and Collective Tasks*

Giorgio Audrito[1][0000−0002−2319−0375], Roberto Casadei[2][0000−0001−9149−949X], Ferruccio Damiani[1][0000−0001−8109−1706], Gianluca Torta[1][0000−0002−4276−7213], and Mirko Viroli[2][0000−0003−2702−5702]

[1] Università di Torino, Italy
{giorgio.audrito,ferruccio.damiani,gianluca.torta}@unito.it
[2] Università di Bologna, Italy
{roby.casadei,mirko.viroli}@unibo.it

Abstract. Recent trends like the Internet of Things (IoT) suggest a vision of dense and multi-scale deployments of computing devices in nearly all kinds of environments. A prominent engineering challenge revolves around programming the collective adaptive behaviour of such computational ecosystems. This requires abstractions able to capture concepts like ensembles (dynamic groups of cooperating devices) and collective tasks (joint activities carried out by ensembles). In this work, we consider collections of devices interacting with neighbours and that execute in nearly-synchronised sense–compute–interact rounds, where the computation is given by a single control program. To support programming whole computational collectives, we propose the abstraction of a distributed collective process (DCP), which can be used to define at once the ensemble formation logic and its collective task. We implement the abstraction in the eXchange Calculus (XC), a core language based on neighbouring values (maps from neighbours to values) where state management and interaction is handled through a single primitive, `exchange`. Then, we discuss the features of the abstraction, its suitability for different kinds of distributed computing applications, and provide a proof-of-concept implementation of a wave-like process propagation.

Keywords: collective computing · collective processes · ensembles · formation control

1 Introduction

Programming the collective behaviour of large collections of computing and interacting devices is a major research challenge, promoted by recent trends like the Internet of Things [6] and swarm robotics [15]. This challenge is investigated

* This publication is part of the project NODES which has received funding from the MUR – M4C2 1.5 of PNRR with grant agreement no. ECS00000036. The work was also partially supported by the Italian PRIN project "CommonWears" (2020HCWWLP) and the EU/MUR FSE PON-R&I 2014-2020.

and addressed by several related research threads including *coordination* [24, 42], *multi-agent systems* [14], *collective adaptive systems* [23], *macroprogramming* [18, 40], *spatial computing* [13], *field-based coordination* [36], *aggregate computing* [49], and *attribute-based communication* [1].

This activity can be supported by suitable *programming abstractions* supporting declarative specifications of collective behaviours. Examples of abstractions include ensembles [41], computational fields [49], collective communication interfaces [50, 1], and collective orchestration tasks [46]. In this work, we cover the abstraction of a *distributed collective process (DCP)*, inspired by aggregate processes [20, 21, 48], which consists of a model for concurrent collective tasks running and spreading on dynamic domains of devices. We provide an abstract model of the abstraction on event structures, and discuss its implementation on the eXchange Calculus (XC) [9], a language, inspired by field calculi [49], for programming homogeneous systems of neighbour-interacting devices. Then, we discuss how the DCP abstraction can support multiple patterns of collective behaviour and self-organisation.

The paper is organised as follows. Section 2 provides context, related work, and motivation. Section 3 reviews the basics of the XC language. Section 4 provides the contribution. Section 5 discusses features and applications of the approach, and provides a proof-of-concept implementation. Section 6 summarises results and points out directions for future work.

2 Context, Related Work, and Motivation

This work lies in the context of models and languages for programming collective behaviour [23, 17, 15]. Indeed, achieving the desired collective behaviour is an engineering goal for different domains and applications:

- *Swarm robotics.* Multiple robots may be tasked to move and act as a collective to explore an unknown environment [37], to search and rescue victims for humanitarian aid after disasters [5], to map a crop field for the presence of weeds [3], to transport objects exceeding [26], etc.
- *The IoT.* The *things* should coordinate to promote application-level goals (e.g., by gathering and processing relevant data) while making efficient use of resources. For instance, the nodes could support the aggregation of machine learning models [47], or collaborate to measure the collective status of the network to support various activities from environment sensing [35] to remote attestation of system integrity [4].
- *Hybrid Collective Intelligence (CI).* Socio-technical systems involving humans and computing devices could be programmed as "social machines" [30] executing coordinated tasks [46], or promoting the emergence of collective knowledge [27].
- *Computing ecosystems.* Modern infrastructures spanning the edge–fog–cloud layers can be considered as collective systems. The computing nodes should exchange and process information to create suitable topologies and structures [33, 45] and coordinate task allocation [38], resiliently.

This problem is at the heart of several related research threads. The field of *coordination* [24, 42] addresses it by governing interaction; *collective adaptive systems* engineering [23] investigates means for collective adaptation in large populations of agents; *spatial computing* [13] leverages spatial abstractions to guide behaviour and perform computation; *macroprogramming* [18, 40] takes a programming language-perspective to expressing macroscopic behaviour; *multi-agent systems* programming [14] considers autonomy, cognitive, and organisational concerns; and so on.

In this work, we consider a language-based software engineering perspective [29]. In other words, we seek for abstractions supporting expressing collective behaviour. Examples of abstractions proposed in previous research include:

- *ensembles* [41]: dynamic composites of devices, e.g., formed by attribute-based formation rules;
- *computational fields* [36, 49]: maps from devices to values, used to capture collective inputs, and collective outputs;
- *aggregate computations* [49]: functions mapping input computational fields to output computational fields, implicitly handling coordination;
- *aggregate processes* [10, 11, 20, 21]: dynamic *aggregate computations* [49] on evolving domains of devices;
- *collective communication interfaces* [50]: abstractions able to flexibly express the targets of communications actions, e.g., via attributes [41, 1];
- *collective-based tasks* [46]: abstractions keeping track of the lifecycle and state of tasks assigned to collectives.

In particular, we consider *collective systems*, namely largely homogeneous collections of devices or agents. Each device can be thought of as a resource that provides capabilities and provides access to a local context that depends on its situation on the environment and possibly its state. For instance, in a smart city, fixed smart lights may be located nearby streets, smart cameras may support monitoring of facilities, smart vehicles may move around to gather city-wide infrastructural data, etc. Since we would like to avoid bottlenecks and single-points-of-failure, we avoid centralisations and opt for a fully decentralised approach: a device can generally interact only within its local context, which may include a portion of the environment and other nearby devices. If our goal is to exploit the distributed, pervasive computer made of an entire collection of situated devices, an idea could be to run *collaborative* tasks involving subsets of devices—to exploit their resources, capabilities, and contexts. Since a process may not know beforehand the resources/capabilities it needs and the relevant contexts, it may embed the logic to look for them, i.e., to spread over the collective system until its proper set of supporting devices have been identified. Moreover, the requirements of the process may change over time, dynamically self-adapting to various environmental conditions and changing goals. Within a process that concurrently spans a collection of devices, local computations may be scheduled and information may flow around in order to support collective activities [15, 53] such as collective perception [28], collective decision-making [16],

collective movement [39], etc. So, if the collective that sustains the process decides that more resources are needed, the process may spread to a larger set of devices; conversely, if the collective task has been completed, the devices may quit the process, eventually making it vanish. This is, informally, our idea of a *distributed collective processes (DCP)*, i.e., a *process* (i.e., a running program or task) which is *collective* (i.e., a joint activity carried out by largely homogeneous entities) and *distributed* (i.e., concurrently active on distinct networked devices), whereby the collective task and the underlying ensemble can mutually affect each other, and ensemble formation is driven by decentralised device-to-device interaction.

In the following, we explain a formal framework (Section 3) particularly suitable to study and implement this DCP abstraction; then, we formalise a language construct (Section 4) to effectively *program* such DCPs; and finally discuss features and applications enabled by our abstraction implementation (Section 5).

3 Background: the eXchange Calculus

We consider the *eXchange Calculus (XC)* [9] as the formal framework for modelling, reasoning about, and implementing DCPs. In this section, we first present the system and execution model (Section 3.1), providing an operational view of the kinds of systems we target, and then describe the basic constructs of XC that we leverage in this work (Section 3.2).

3.1 System Model

The target system that we would like to program can be modelled as a collection of *nodes*, able to interact with the environment through *sensors* and *actuators*, and able to communicate with *neighbours* by exchanging messages. We assume that each node runs in asynchronous *sense–compute–act rounds*, where

1. *sense*: the node queries sensors for getting up-to-date environmental values, and gathers recent messages from neighbours (which may expire after some time)—all this information is what we call as the node's *context*;
2. *compute*: the node evaluates the common control program, mapping the context (i.e., inputs from sensors and neighbours) to an output describing the actions to perform (i.e., actuations and communications);
3. *act*: the node executes the actions as dictated by the program, possibly resulting into environment change or message delivery to neighbours.

This kind of loop is used to ensure that the context is continuously assessed (at discrete times), and the reactions are also computed and performed continuously. This model has shown to be particularly useful to specify self-organising and collective adaptive behaviours, especially for long-running coordination tasks [49].

The semantic of a system execution can be expressed as an event structure (see Figure 1), where events ϵ denote whole sense–compute–act rounds, and arrows between events denote that certain source events have provided inputs

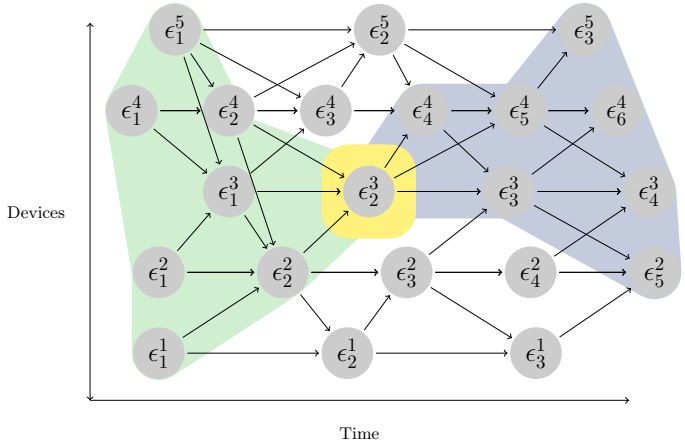

Fig. 1. Example of an event structure modelling a distributed system execution. Nodes labelled by ϵ_k^δ denote the k-th round of device δ. The yellow background highlights a reference event, from which its past (green) and future (blue) are identified through the causal relationship implied by the arrows denoting neighbour events.

(i.e., messages) to target events. In particular, if event ϵ' is connected with an arrow to ϵ, we say that ϵ' is a *neighbour* of ϵ, denoted $\epsilon' \rightsquigarrow \epsilon$. We denote with $N(\epsilon)$ the set of all *neighbours* of ϵ, and with $d(\epsilon)$ the device where event ϵ happens, i.e., where it is executed.

Programming the systems described in this section thus means defining the control rules that specify how the context at each event is mapped to the messages to be sent to neighbour events.

3.2 XC key data type: Neighbouring Values

In XC we distinguish two types of values. The *Local* values ℓ include classic atomic and structured types A such as int, float, string or list. The neighbouring values (*nvalues*) are instead maps $\underline{\mathtt{w}}$ from device identifiers δ_i to corresponding local values ℓ_i, with an additional local value ℓ that acts as a *default*:

$$\underline{\mathtt{w}} = \ell[\delta_1 \mapsto \ell_1, \dots, \delta_n \mapsto \ell_n]$$

A nvalue specifies a (set of) values received from or sent to neighbours: received values are gathered into nvalues, then can be locally processed, and the final resulting nvalue can be interpreted as messages to be sent back to neighbours. The devices with an associated entry in the nvalue are thus typically a (small) subset of all devices, namely those that are close-enough to the current device, and which are of course working correctly.

The default is used when a value is not available for some neighbour δ', e.g., because δ' has just been switched on and has not yet produced a value, or because it has just moved close enough to the current device δ to become

one of its neighbours. The notation above should therefore read as "the nvalue $\underline{\mathtt{w}}$ is ℓ everywhere (i.e., for all neighbours) except for devices $\delta_1, \ldots, \delta_n$ which have values $\ell_1, \ldots, \ell_n$.

To exemplify nvalues, in Figure 1, upon waking up for computation ϵ_2^3, device δ_3 may process a nvalue $\underline{\mathtt{w}} = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2, \delta_2 \mapsto 3]$, corresponding to the messages carrying scalar values 1, 2, and 3 received when asleep from δ_4, δ_3 (i.e., *itself* at the previous round), and δ_2. The entries for all other (neighbour) devices default to 0. After the computation, δ_2 may send out the messages represented by the nvalue $\underline{\mathtt{w}}' = 0[\delta_4 \mapsto 5, \delta_3 \mapsto 6]$; so that 5 is sent to δ_4, 6 is sent to δ_3, and 0 is sent to every other device, such as a newly-connected device. For convenience, we may use the notation $\underline{\mathtt{w}}(\delta')$ for the local value (specific or default) associated with δ' by $\underline{\mathtt{w}}$.

Note that a local value ℓ can be naturally converted to a nvalue $\ell[]$ where it is the default value for every device. Except for clarity, thus local values and nvalues can be treated uniformly. Functions on local values are implicitly lifted to nvalues, by applying them on the maps' content pointwise. For example, if $\underline{\mathtt{w}}_1$ assigns value 2 to δ_3 and $\underline{\mathtt{w}}_2$ assigns default value 1 to δ_3, then $\underline{\mathtt{w}}_3 = \underline{\mathtt{w}}_1 \cdot \underline{\mathtt{w}}_2$ shall assign value $2 \cdot 1 = 2$ to δ_3.

A fundamental operation on nvalues is provided by the built-in function $\mathtt{nfold}(f : (A, B) \to A, \underline{\mathtt{w}} : \underline{B}, \ell : A) : A$. As suggested by the name, the function folds over a nvalue $\underline{\mathtt{w}}$, i.e., starting from a base local value ℓ it repeatedly applies function f to neighbours' values in $\underline{\mathtt{w}}$, excluding the value for the current device. For instance, if δ_2 with set of neighbours $\{\delta_1, \delta_3, \delta_4\}$ performs a `nfold` operation $\mathtt{nfold}(*, \underline{\mathtt{w}}, 1)$, the output will be $1 \cdot \underline{\mathtt{w}}(\delta_1) \cdot \underline{\mathtt{w}}(\delta_3) \cdot \underline{\mathtt{w}}(\delta_4)$. Note that, as nvalues are *unordered maps*, it is sensible to assume that f is associative and commutative.

Two built-in operations on nvalues act on the value associated with the current (self) device:

- $\mathtt{self}(\underline{\mathtt{w}} : \underline{A}) : A$ returns the local value $\underline{\mathtt{w}}(\delta)$ in $\underline{\mathtt{w}}$ for the self device δ
- $\mathtt{updateSelf}(\underline{\mathtt{w}} : \underline{A}, \ell : A) : \underline{A}$ returns a nvalue where the value for the self device δ is set to ℓ.

There are several other fundamental built-in operators in XC, such as `exchange` and `mux`, which are however not necessary for understanding the rest of this paper. Please refer to [9] for their detailed, formal description.

4 Distributed Collective Processes in XC

In this section, we present an implementation of DCPs in XC. First, we characterise the implementation in terms of an abstract notation and formulas on event structures (Section 4.1). Then, we provide a formalisation of DCPs in terms of the big-step operational semantics for a new XC construct (Section 4.2), that can be used to actually *program* DCPs.

4.1 Modelling on event structures

In this discussion, we refer to an event structure as the one depicted in Figure 1. A *distributed collective process (DCP)* P is a computation with given programmed behaviour. A single DCP can be run in multiple *process instances* P_i, each associated to a unique *process identifier (PID)* i, which we assume also embeds construction parameters for the process instance. New instances of an aggregate process P are spawned through a *generation field* G_P, producing a set of identifiers $G(\epsilon) = \{i \dots\}$ in each event ϵ, of process instances that need to be created in that event ϵ (which we call *initiator* for P_i). For each process instance P_i, we use the Boolean predicate $\pi_{P_i}(\epsilon)$ to denote whether such instance is being executed at event ϵ (either being initiated by ϵ, or through propagation from previous events). Each process instance P_i, if active in an event ϵ (i.e., $\pi_{P_i}(\epsilon) = \top$), locally computes both an *output* $O_{P_i}(\epsilon)$ (returned to the process caller) and a *status* $s_{P_i}(\epsilon)$, which is an nvalue mapping the device d of each neighbour event $\epsilon \in N$ to a `bool` value.

A process instance P_i which is active in an event ϵ *potentially* propagates the process to any event ϵ' of which ϵ is a neighbour ($\epsilon \rightsquigarrow \epsilon'$) depending on the value of $s_{P_i}(\epsilon)$. In formulas:

$$\pi_{P_i}(\epsilon) = \begin{cases} \top & \text{if } i \in G_P(\epsilon) \\ \top & \text{if } \exists \epsilon' \rightsquigarrow \epsilon.\ \pi_{P_i}(\epsilon') \wedge s_{P_i}(\epsilon')(d(\epsilon)) = \top \\ \bot & \text{otherwise.} \end{cases}$$

The XC defines a built-in construct $spawn_{XC}(P, G_P)$ that runs independent instances of a field computation P, where new instances are locally generated according to *generation field* G_P as explained above. The output of a $spawn_{XC}(P, G_P)$ expression in an event ϵ is the set of pairs $\{(i, O_{P_i}(\epsilon)), \dots\}$ for which $\pi_{P_i}(\epsilon) = \top$.

4.2 Formalisation

The $spawn_{XC}$ construct, defined mathematically in the previous Section 4.1, embeds naturally in XC as a built-in function, derived by converting the classical spawn construct [20] into XC. As a built-in in XC, `spawn` assumes the same type as the classical spawn construct in field calculus: $\forall \alpha_k, \alpha_v.((\alpha_k) \rightarrow \texttt{pair}[\alpha_v, \texttt{bool}], \texttt{set}[\alpha_k]) \rightarrow \texttt{map}[\alpha_k, \alpha_v]$. However, in XC every type allows nvalues, which translates into practical differences.

Figure 2 presents the semantics of the spawn built-in, relative to the XC semantics presented in [9], which we do not include for brevity. As in [32], the overbar notation indicates a (possibly empty) sequence of elements, and multiple overbars are expanded together, e.g., $\overline{x} \mapsto \overline{y}$ is short for $x_1 \mapsto y_1, \dots, x_n \mapsto y_n$ ($n \geq 0$). The semantics is given by the auxiliary evaluation judgement for built-ins $\delta; \sigma; \Theta \vdash \texttt{b}(\overline{\texttt{w}}) \Downarrow^* \texttt{w}; \theta$, to be read as "expression $\texttt{b}(\overline{\texttt{w}})$ evaluates to nvalue $\texttt{w}$ and value-tree θ on device δ with respect to sensor values σ and value-tree environment Θ", where:

Auxiliary definitions:

$\theta ::= \langle \overline{\theta} \rangle \mid \mathtt{w}\langle \overline{\theta} \rangle \mid \overline{\ell} \mapsto \overline{\theta}$ value-tree

$\pi^{\ell}(\overline{\ell} \mapsto \overline{\theta}) = \theta_i$ s.t. $\ell_i = \ell$ if it exists else $\bullet$

Auxiliary evaluation rules: $\boxed{\delta; \sigma; \Theta \vdash \mathtt{f}(\overline{\mathtt{w}}) \Downarrow^* \mathtt{w}; \theta}$

$$[\text{A-SPAWN}]\ \frac{k_1, \ldots, k_n = \mathtt{w}^k(\delta) \cup \{k \text{ for } \delta' \in \mathbf{dom}(\Theta), k \mapsto b\langle\theta\rangle \in \Theta(\delta') \text{ with } b(\delta) = \mathtt{True}\} \quad \delta; \sigma; \pi_1(\pi^{k_i}(\Theta)) \vdash \mathtt{w}^p(k_i) \Downarrow \mathtt{w}_i; \theta_i \text{ where } \mathtt{w}_i = \mathtt{pair}(\mathtt{v}_i, b_i) \text{ for } i \in 1, \ldots, n}{\delta; \sigma; \Theta \vdash \mathtt{spawn}(\mathtt{w}^p, \mathtt{w}^k) \Downarrow^* \overline{k} \mapsto \overline{\mathtt{w}}; \overline{k} \mapsto \overline{b}\langle\overline{\theta}\rangle}$$

Fig. 2. Device (big-step) operational semantics of FXC

- θ is an ordered tree with nvalues on some nodes, representing messages to be sent to neighbours by tracking necessary nvalues and stack frames produced while evaluating $\mathtt{b}(\overline{\mathtt{w}})$;
- Θ collects the most recent value-trees received by neighbours of δ, as a map $\delta_1 \mapsto \theta_1$, ..., $\delta_n \mapsto \theta_n$ ($n \geq 0$) from device identifiers to value-trees.

In order to introduce the `spawn` construct, it is necessary to extend the auxiliary definition of value-trees (highlighted in grey), to also allow for maps from local literals ℓ (identifiers of the running processes) to their corresponding value-trees. Then, rule [A-SPAWN] can be written by naturally porting the similar rule in [20], while using the fact that the Boolean returned by the process is an nvalue, and thus can be different for different neighbours. In this rule, a list of *process keys* $\overline{k}$ is computed by adjoining *(i)* the keys $\mathtt{w}^k(\delta)$ currently present in the second argument $\mathtt{w}^k$ of `spawn` for the current device δ; *(ii)* the keys that any neighbour δ' broadcast in their last message $\Theta(\delta')$, provided that the corresponding Boolean value b returned was true for the current device $b(\delta) = \mathtt{True}$ (thus, demanding process expansion to δ). To realise "multiple alignment", for each key k_i, the process $\mathtt{w}^p$ is applied to k_i with respect to the part of the value-tree environment $\pi_1(\pi^{k_i}(\Theta))$ that corresponds to key k_i, producing $\mathtt{w}_i; \theta_i$ as a result. Finally, the construct concludes returning the maps $\overline{k} \mapsto \overline{\mathtt{w}}; \overline{k} \mapsto \overline{\theta}$ mapping process keys to their evaluation result.

5 Discussion and Proof-of-Concept

In this section, we discuss the proposed abstraction (Section 5.1), the characteristics of the proposed programming model for DCPs (Section 5.2), then provide examples of applications (Section 5.3), and provide a proof-of-concept implementation of a wave-like propagation of a DCP (Section 5.4).

5.1 The DCP abstraction

The crucial problem that we investigate in this paper revolves around the definition of collaborative activities carried out by *dynamic* collections of devices

(a.k.a. *ensembles* [41]). We call these *DCPs* since they are defined by a common control program that regulates the behaviour of a largely homogeneous set of devices.

In particular, a device may participate *concurrently* to multiple collective processes, or to multiple ensembles. How participation to multiple DCPs relates to local resource usage (cf. resource-constrained devices) is abstracted away and may be dealt both programmatically (e.g., through a status computed depending on the resource availability perceived through a sensor) or automatically at the virtual machine level (e.g., by runtime checks). Furthermore, notice that, in any single round, a device executes the computation associated to all the currently joined DCPs. That is, in the basic model, the number of processes joined by a device has no effect whatsoever on the number of rounds, which follow a given scheduling policy. Therefore, the participation to several processes may in principle increase the duration of rounds significantly, possibly slowing down the reactivity of a device; so, real-world implementations have also to consider these aspects. Associating different scheduling policies to different processes is however possible, but requires an extension to the basic execution model, e.g., along the lines of [44].

We define as the *domain* of a DCP the set of the nodes that are currently running it. We define as the *shape* of a DCP the spatiotemporal region that is identified by the spatiotemporal locations of the nodes belonging to the domain of the DCP. Often, DCPs are *transient*, i.e., they have a limited lifetime: they start to exist at some time, and they dissolve once no more nodes run them.

In this work, we are mainly concerned with studying how to create and manipulate these DCPs. The supporting formal framework and implementation is described in Section 4. In summary, the developer has the following mechanisms for defining systems of DCPs:

- *generation logic*: the need for collective activities can be encoded in a rule for generating new instances of DCPs;
- *identity logic*: the logic used to identify DCPs can be used to distinguish between them and hence to regulate their domains (e.g., for controlling the granularity of teams);
- *internal logic*: this logic defines a collective computation (scoped within the domain of a single DCP) promoting decentralised decision-making, e.g., in terms of typical self-organisation patterns (collection, propagation, leader election, evaporation, etc.);
- *shape control logic*: it is possible to specify rules for the local expansion of the domains of DCPs (e.g., to gather more participants), typically also leveraging results from the internal computation itself—the XC implementation provides an especially flexible way to specify this, as different neighbours can receive different information;
- *termination logic*: this logic, strictly related to shape control, enables to specify how individual agents may leave a DCP instance as well as how an entire DCP may be terminated;
- *input logic*: existing DCPs may also be controlled be specifying "external inputs" provided as explicit arguments or closed over a lambda closure—an

example is *meta-control logic*, based on inspecting the (results of) multiple DCPs to take decisions about their evolution.

In the following, we discuss features and examples of use of the abstraction.

5.2 Features of the abstraction and programming model

Progressive and live construction of ensembles (cf. self-organisation, self-healing, etc.). The DCPs have a *dynamic* domain, that evolves progressively to include more or less devices. The devices at the border of the DCP can choose to expand it to (a subset of) their neighbours and the neighbours themselves can opt in or out. Moreover, since evaluation of the program is repeated over time, the border is *live*, meaning that membership can be always re-evaluated, in order to consider the up-to-date context. Conversely, members that are no longer interested in participating in the collective task, or that have completed the tasks associated to their role, can leave the process by returning `False`[] in the spawn routine, or even start process termination patterns as those investigated in [11].

Flexible control of collective process shape and state. The shape and state of a DCP can be regulated flexibly, by leveraging different kinds of mechanisms. For instance, the state and shape of a process can be controlled at a collective level, as a result of a collective consensus or decision-making activity. As an alternative, the leader or owner of the DCP may centralise some of the decision-making: for instance, it may gather statistics from its members (using adaptive collection algorithms [8]), and use locally-computed policies to decide whether to let more members join (sharing the local decision with a resilient broadcast algorithm [49]). Between fully centralised and fully decentralised settings, there are intermediate solutions based e.g. on a partitioning of the DCP into sub-groups using partitioning mechanisms that can be applied at the aggregate programming level [2, 19]. The state can be used, for instance, to denote different *phases* of a collective task [22], hence it is important that all the members of the DCP eventually become aware of the up-to-date situation. Regarding shape control, further flexibility is provided by XC, thanks to differentiated messages to neighbours: this feature could be used to essentially control the direction of process propagation (e.g., by filtering, random selective choice, or any other ad-hoc mechanisms).

Support for privacy-preserving collective computations. The possibility in XC to send differentiated messages to neighbours (unlike classical field calculi [49]), especially when supported infrastructurally through point-to-point communication channels, can also promote *privacy* in collective computations. This way, devices that are unrelated to certain tasks, are not exposed to the information that those tasks are being carried out.

5.3 Examples

Given its features, the DCP abstraction could turn useful to program several kinds of higher-level distributed computing abstractions and tasks such as, for instance, the following.

Modelling of teams or ensembles of agents [41]. A DCP can represent, through its very domain, the set of agents that belong to a certain team or ensemble. It can spread around the system to gather and (re-)evaluate a membership condition, to effectively recruit agents into different organisational structures [31]. Two main mechanisms regulate the joining of devices into DCPs: the propagation of PIDs to neighbours (i.e., an *internal* control of the process border), and the possibility to *leave* a process by a device that received a PID, which would not propagate the process further (i.e., an *external* control of the process border). The former mechanism is directly supported by our XC implementation, through the notion of *differentiated messages* to neighbours, enabled by nvalues. Concurrent participation to multiple teams is directly supported by the fact that a single device can participate in an arbitrary number of DCPs. As participation to multiple DCPs leads to increased resource requirements (in both computation time and message size), the programmer has to take into account performance issues when designing the generation and propagation logic of concurrent DCPs. However, the fully asynchronous and resilient nature of XC implies that some additional slack can be used on top of resource bounds posed by the architecture, as longer round execution or message exchange time (or even a device crash) can be handled seamlessly by the XC programming model. Last but not least, the activity within a DCP can be used to support the coordination *within* the ensemble it represents, e.g., through gossip or information spreading algorithms, whose scope is limited to the domain of the DCP; therefore, it may be useful also for *privacy-preserving* computations.

Space-based coordination (e.g., spatiotemporal tuples [22]). A DCP could also be attached to a spatial location—to implemented *spatially-attached processes*. This could be used to support space-based coordination, or to implement coordination models like *spatiotemporal tuples* [22], whereby tuples and tuple operations can be emitted to reside at or query a particular spatial location. To implement the spatiotemporal tuples model, an DCP instance can be used to represent a single `out` (writing), `rd` (reading), and `in` (retrieval) operation—see Figure 3 for a visual example. A tuple is denoted by its `out` process: it exists as long as its DCP is alive in some device. Creating tuples that reside at a fixed spatial location/area (e.g., as described by geodetic coordinates) or that remain attached to a particular mobile device is straightforward. In the former case, the DCP membership condition is just that the device's current location is inside or close by the provided spatial location. In the latter case, the DCP membership condition is just that the device's current distance to the DCP source device (which may be computed by a simple gradient) is within a certain threshold. We may call these *node-attached processes*: as a node moves, a DCP attached to it can follow through, to support collective contextual services; for instance, a node may recruit other nodes and resources for mobile tasks.

Creation of adaptive system structures to support communication and coordination. The ability of DCPs to capture both the formation evolution and the collective activity of a group of devices within a pervasive computing system can

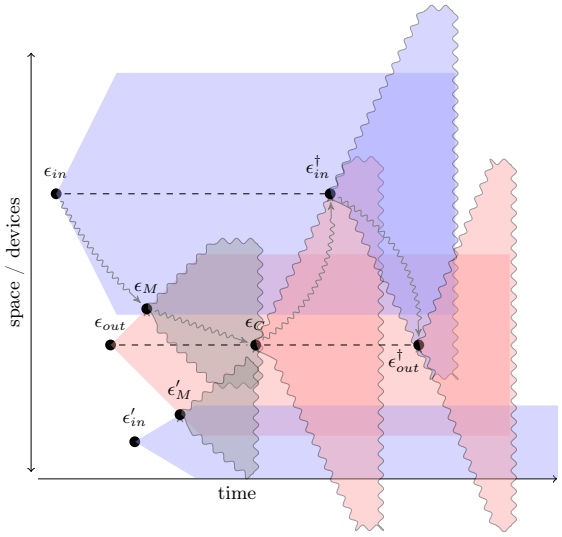

Fig. 3. Graphics of interacting DCPs modelling spatiotemoral tuple operations. Each DCP is denoted as a trapezoid-like shape that springs out at a certain event (a round by a single device). Notation: ϵ_{in} and ϵ_{out} mean that the event generates a process modelling an `in` (retrieval) and `out` (writing) tuple operation, respectively; ϵ_M means that a matching `out` tuple for an `in` operation has been found; ϵ_C means that the `out` tuple process has reached consensus about the `in` process to serve; the † superscript denotes a termination event, starting a process to close an existing process.

be leveraged to create resilient structures supporting non-local or system-wide coordination. For instance, this can be used to implement messaging channels in a peer-to-peer network of situated neighbour-interacting devices (e.g., in a smart city) [20]: the channel consists only of the devices between the source and the destination of a message, hence avoiding expensive gossip or flooding processes that would (i) consume resources of possibly all the devices in the system, and (ii) exacerbate privacy and security issues. As another example, consider the *Self-organising Coordination Regions (SCR)* pattern [19]: it is a mechanism to control the level of decentralisation in systems to flexibly support situated tasks, based on (i) decentralised leader election [43]; (ii) creation of areas around the leaders to basically partition the overall system into manageable regions [51]; and (iii) supporting intra-region and inter-region coordination e.g. by means of *information flows* [52]. Now, traditional solutions based on field calculi [19] do not easily allow for the partitions to *overlap*: this, instead, could be desired for fault-tolerance, flexibility, and improved interaction between adjacent regions, and it turns out to be easily implementable using DCPs.

Modelling of epidemic processes [25]. Finally, DCP also represent a tool for studying how to relate computation, coordination, and epidemic processes. By

the ability of programmatically controlling how processes spread, possibly using conditions that depend on the collective computation carried out by the current ensemble, it is possible to model complex diffusion dynamics. In future work, it would be interesting to explore how XC programs leveraging DCPs could promote network-based and agent-based simulation models for epidemic spread, e.g. such as those reviewed in [34].

5.4 Proof-of-Concept Implementation

As a proof-of-concept of the techniques described in this paper, we have implemented a simple use case exploiting the FCPP simulator [7, 12], which has been extended to support the XC and, in particular, the $spawn_{XC}$ built-in construct described above. The implemented use case is a network of devices where, at some point in time, a source device δ_{FROM} sends a message through a DCP to reach a destination device δ_{TO}. For simplicity, we considered devices to be stationary in the simulation, thus inducing a fixed network topology (although the proof-of-concept program could be run with dynamic topologies as well). Round durations are not identical (they can vary by 10% from base value).

We implement a *spherical* propagation, where the message originating in δ_{FROM} spreads radially in 3D trying to reach δ_{TO}. The process function executed by each process node implements the following logic:

1. if the self device δ is δ_{TO} just return a *false* nvalue $F[]$ (no propagation, since destination has been reached);
2. if it is the first round that δ executes the process, and the neighbours it knows (i.e., that have propagated the process to δ) are $\delta_1, \ldots, \delta_k$, it propagates it to itself and to its *new neighbours* (that are not yet in the process) by returning an nvalue $T[\delta \mapsto T, \delta_1 \mapsto F, \ldots]$;
3. finally, at the second round δ exits itself the process by returning a *false* nvalue $F[]$.

The following snippet of FCPP code shows the core of the simple function described above:

```
1        if (dest)
2            fdwav = field<bool>(false);
3        else if (rnd == 1) {
4            fdwav = field<bool>(false);
5            fdwav = mod_self(CALL, fdwav, true);
6            fdwav = mod_other(CALL, fdwav, true);
7        } else
8            fdwav = field<bool>(false);
```

Note that: flag `dest` is true only on device δ_{TO}; `fdwav` if the field that determines process propagation; a call to `field<bool>(false)` constructs a constant field of Booleans set to `false`; and a call to `mod_self` (resp. `mod_other`) sets the value in a field for the current device (resp. its known neighbours).

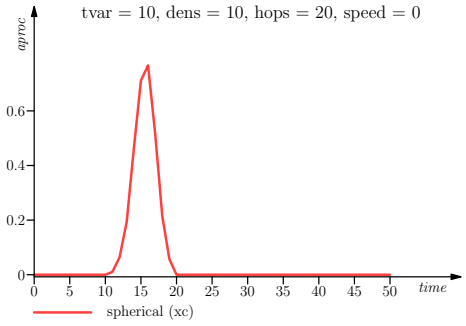

Fig. 4. Average number of active processes over time for the single-process use case.

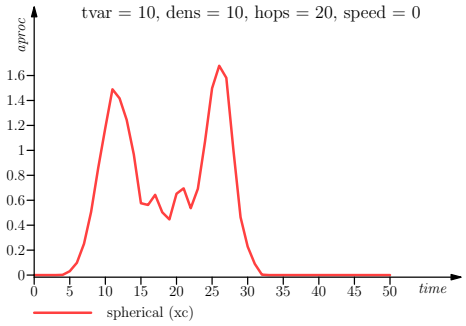

Fig. 5. Average number of active processes over time for the multi-process use case.

We exploit FCPP to simulate a first use case where only one process is generated. The simulation shows that the process propagates *as a wave* starting from δ_{FROM} outwards. Immediately after the wave front goes beyond a device, the device itself exits the process thus releasing potentially precious resources for other computations. Figure 4 shows the average number of active processes (*aproc*) within the network of devices in one specific execution of the use case (which took a time interval $[1, 50]$). For the first 10 sec, the average is 0, since no process has been created yet. After the process is created by δ_{FROM}, it propagates until it reaches its destination, and then quickly vanishes.

In a second use case, we let 10 different devices generate a new process at each round with probability 5%, in the time interval $[1, 25]$. Figure 5 shows that the average number of active processes *aproc* grows from 0 (at time 0) up to slightly more than 1.6 (just after 25), and then quickly drops again down to 0. Given that more than 10 processes are generated during the use case, the average number of active processes is kept low by the fact that the nodes exploit $spawn_{XC}$ to immediately exit processes after entering and propagating them.

6 Conclusion

In this paper, we have covered the abstraction of a *distributed collective process (DCP)*, which supports the definition of the collective adaptive behaviour of pervasive collections of neighbour-interacting devices working in sense–compute–interact rounds. In particular, DCPs model decentralised collective tasks that also move, spread, and retract over the collective system in which they are spawned. We have discussed the abstraction, analysed it in the general framework of event structures, and implemented it in the *eXchange Calculus (XC)*, a minimal core language particularly suitable for implementing DCPs, for its *Neighbouring Value* data structure, that enables fine-tuned control of what data gets shared with neighbours. Finally, we have discussed its features and applicability, and have shown a proof of concept implementation of a wave-propagation algorithm—which may be used for model resource-efficient information flows.

In future work, we would like to further study DCPs by a dynamical perspective, and possibly explore its ability to model and simulate epidemic processes. Further, we would like to implement in XC a library of reusable functions capturing common patterns of DCPs usage, to streamline pervasive and collective computing application development.

References

1. Abd Alrahman, Y., De Nicola, R., Loreti, M.: Programming interactions in collective adaptive systems by relying on attribute-based communication. Science of Computer Programming **192** (2020). https://doi.org/10.1016/j.scico.2020.102428
2. Aguzzi, G., Audrito, G., Casadei, R., Damiani, F., Torta, G., Viroli, M.: A field-based computing approach to sensing-driven clustering in robot swarms. Swarm Intell. **17**(1), 27–62 (2023). https://doi.org/10.1007/s11721-022-00215-y
3. Albani, D., IJsselmuiden, J., Haken, R., Trianni, V.: Monitoring and mapping with robot swarms for agricultural applications. In: 14th IEEE International Conference on Advanced Video and Signal Based Surveillance, AVSS 2017, Lecce, Italy, August 29 - September 1, 2017. pp. 1–6. IEEE Computer Society (2017). https://doi.org/10.1109/AVSS.2017.8078478, https://doi.org/10.1109/AVSS.2017.8078478
4. Ambrosin, M., Conti, M., Lazzeretti, R., Rabbani, M.M., Ranise, S.: Collective remote attestation at the internet of things scale: State-of-the-art and future challenges. IEEE Commun. Surv. Tutorials **22**(4), 2447–2461 (2020). https://doi.org/10.1109/COMST.2020.3008879

5. Arnold, R., Jablonski, J., Abruzzo, B., Mezzacappa, E.: Heterogeneous UAV multi-role swarming behaviors for search and rescue. In: Rogova, G., McGeorge, N.M., Ruvinsky, A., Fouse, S., Freiman, M.D. (eds.) IEEE Conference on Cognitive and Computational Aspects of Situation Management, CogSIMA 2020, Victoria, BC, Canada, August 24-29, 2020. pp. 122–128. IEEE (2020). https://doi.org/10.1109/CogSIMA49017.2020.9215994
6. Atzori, L., Iera, A., Morabito, G.: The internet of things: A survey. Comput. Networks **54**(15), 2787–2805 (2010). https://doi.org/10.1016/j.comnet.2010.05.010
7. Audrito, G.: FCPP: an efficient and extensible field calculus framework. In: International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). pp. 153–159. IEEE (2020). https://doi.org/10.1109/ACSOS49614.2020.00037
8. Audrito, G., Casadei, R., Damiani, F., Pianini, D., Viroli, M.: Optimal resilient distributed data collection in mobile edge environments. Comput. Electr. Eng. **96**(Part), 107580 (2021). https://doi.org/10.1016/j.compeleceng.2021.107580
9. Audrito, G., Casadei, R., Damiani, F., Salvaneschi, G., Viroli, M.: Functional programming for distributed systems with XC. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany. LIPIcs, vol. 222, pp. 20:1–20:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.20
10. Audrito, G., Casadei, R., Torta, G.: Towards integration of multi-agent planning with self-organising collective processes. In: IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2021, Companion Volume, Washington, DC, USA, September 27 - Oct. 1, 2021. pp. 297–298. IEEE (2021). https://doi.org/10.1109/ACSOS-C52956.2021.00042
11. Audrito, G., Casadei, R., Torta, G.: On the dynamic evolution of distributed computational aggregates. In: IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion, ACSOS-C 2022, Virtual, CA, USA, September 19-23, 2022. pp. 37–42. IEEE (2022). https://doi.org/10.1109/ACSOSC56246.2022.00024
12. Audrito, G., Rapetta, L., Torta, G.: Extensible 3d simulation of aggregated systems with FCPP. 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, Lucca, Italy, June 13-17, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13271, pp. 55–71. Springer (2022). https://doi.org/10.1007/978-3-031-08143-9_4
13. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: Languages for spatial computing. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, chap. 16, pp. 436–501. IGI Global (2013). https://doi.org/10.4018/978-1-4666-2092-6.ch016
14. Boissier, O., Bordini, R.H., Hubner, J., Ricci, A.: Multi-agent oriented programming: programming multi-agent systems using JaCaMo. Mit Press (2020)
15. Brambilla, M., Ferrante, E., Birattari, M., Dorigo, M.: Swarm robotics: a review from the swarm engineering perspective. Swarm Intell. **7**(1), 1–41 (2013). https://doi.org/10.1007/s11721-012-0075-2, https://doi.org/10.1007/s11721-012-0075-2
16. Bulling, N.: A survey of multi-agent decision making. KI - Künstliche Intelligenz **28**(3), 147–158 (Jul 2014). https://doi.org/10.1007/s13218-014-0314-3, https://doi.org/10.1007/s13218-014-0314-3

17. Casadei, R.: Artificial collective intelligence engineering: a survey of concepts and perspectives (2023). https://doi.org/10.48550/ARXIV.2304.05147, https://arxiv.org/abs/2304.05147, Accepted for publication in the Artificial Life journal (MIT Press).
18. Casadei, R.: Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling. ACM Computing Surveys (Jan 2023). https://doi.org/10.1145/3579353, https://doi.org/10.1145/3579353
19. Casadei, R., Pianini, D., Viroli, M., Natali, A.: Self-organising coordination regions: A pattern for edge computing. In: Nielson, H.R., Tuosto, E. (eds.) 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, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11533, pp. 182–199. Springer (2019). https://doi.org/10.1007/978-3-030-22397-7_11
20. Casadei, R., Viroli, M., Audrito, G., Pianini, D., Damiani, F.: Aggregate processes in field calculus. In: Nielson, H.R., Tuosto, E. (eds.) 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, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11533, pp. 200–217. Springer (2019). https://doi.org/10.1007/978-3-030-22397-7_12, https://doi.org/10.1007/978-3-030-22397-7_12
21. Casadei, R., Viroli, M., Audrito, G., Pianini, D., Damiani, F.: Engineering collective intelligence at the edge with aggregate processes. Engineering Applications of Artificial Intelligence **97**, 104081 (2021)
22. Casadei, R., Viroli, M., Ricci, A., Audrito, G.: Tuple-based coordination in large-scale situated systems. In: Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12717, pp. 149–167. Springer (2021). https://doi.org/10.1007/978-3-030-78142-2_10
23. De Nicola, R., Jähnichen, S., Wirsing, M.: Rigorous engineering of collective adaptive systems: special section. Int. J. Softw. Tools Technol. Transf. **22**(4), 389–397 (2020). https://doi.org/10.1007/s10009-020-00565-0, https://doi.org/10.1007/s10009-020-00565-0
24. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM **35**(2), 96–107 (1992). https://doi.org/10.1145/129630.376083, https://doi.org/10.1145/129630.376083
25. Giudice, N.D., Matteucci, L., Quadrini, M., Rehman, A., Loreti, M.: Sibilla: A tool for reasoning about collective systems. In: ter Beek, M.H., Sirjani, M. (eds.) Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13271, pp. 92–98. Springer (2022). https://doi.org/10.1007/978-3-031-08143-9_6
26. Groß, R., Dorigo, M.: Towards group transport by swarms of robots. Int. J. Bio Inspired Comput. **1**(1/2), 1–13 (2009). https://doi.org/10.1504/IJBIC.2009.022770

27. Gruber, T.: Collective knowledge systems: Where the social web meets the semantic web. J. Web Semant. **6**(1), 4–13 (2008). https://doi.org/10.1016/j.websem.2007.11.011
28. Gunther, H., Riebl, R., Wolf, L.C., Facchi, C.: Collective perception and decentralized congestion control in vehicular ad-hoc networks. In: 2016 IEEE Vehicular Networking Conference, VNC 2016, Columbus, OH, USA, December 8-10, 2016. pp. 1–8. IEEE (2016). https://doi.org/10.1109/VNC.2016.7835931
29. Gupta, G.: Language-based software engineering. Sci. Comput. Program. **97**, 37–40 (2015). https://doi.org/10.1016/j.scico.2014.02.010
30. Hendler, J., Berners-Lee, T.: From the semantic web to social machines: A research challenge for AI on the world wide web. Artif. Intell. **174**(2), 156–161 (2010). https://doi.org/10.1016/j.artint.2009.11.010
31. Horling, B., Lesser, V.R.: A survey of multi-agent organizational paradigms. Knowl. Eng. Rev. **19**(4), 281–316 (2004). https://doi.org/10.1017/S0269888905000317
32. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems **23**(3), 396–450 (2001)
33. Karagiannis, V., Schulte, S.: Distributed algorithms based on proximity for self-organizing fog computing systems. Pervasive Mob. Comput. **71**, 101316 (2021). https://doi.org/10.1016/j.pmcj.2020.101316
34. Li, J., Xiang, T., He, L.: Modeling epidemic spread in transportation networks: A review. Journal of Traffic and Transportation Engineering (English Edition) **8**(2), 139–152 (Apr 2021). https://doi.org/10.1016/j.jtte.2020.10.003
35. Liu, C., Hua, J., Julien, C.: SCENTS: collaborative sensing in proximity iot networks. In: IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2019, Kyoto, Japan, March 11-15, 2019. pp. 189–195. IEEE (2019). https://doi.org/10.1109/PERCOMW.2019.8730863
36. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications with the TOTA middleware. In: Pervasive Computing and Communications, 2004. pp. 263– 273. IEEE (2004). https://doi.org/10.1109/PERCOM.2004.1276864
37. McGuire, K., Wagter, C.D., Tuyls, K., Kappen, H.J., de Croon, G.C.H.E.: Minimal navigation solution for a swarm of tiny flying robots to explore an unknown environment. Sci. Robotics **4**(35) (2019). https://doi.org/10.1126/scirobotics.aaw9710
38. Mohan, N., Kangasharju, J.: Edge-fog cloud: A distributed cloud for internet of things computations. In: 2016 Cloudification of the Internet of Things, CIoT 2016, Paris, France, November 23-25, 2016. pp. 1–6. IEEE (2016). https://doi.org/10.1109/CIOT.2016.7872914
39. Navarro, I., Matía, F.: A survey of collective movement of mobile robots. International Journal of Advanced Robotic Systems **10**(1), 73 (Jan 2013). https://doi.org/10.5772/54600, https://doi.org/10.5772/54600
40. Newton, R., Welsh, M.: Region streams: Functional macroprogramming for sensor networks. In: Workshop on Data Management for Sensor Networks. pp. 78–87 (2004). https://doi.org/10.1145/1052199.1052213
41. Nicola, R.D., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: The SCEL language. ACM Trans. Auton. Adapt. Syst. **9**(2), 7:1–7:29 (2014). https://doi.org/10.1145/2619998
42. Papadopoulos, G.A., Arbab, F.: Coordination models and languages. Adv. Comput. **46**, 329–400 (1998). https://doi.org/10.1016/S0065-2458(08)60208-9
43. Pianini, D., Casadei, R., Viroli, M.: Self-stabilising priority-based multi-leader election and network partitioning. In: IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2022,

Virtual, CA, USA, September 19-23, 2022. pp. 81–90. IEEE (2022). https://doi.org/10.1109/ACSOS55765.2022.00026
44. Pianini, D., Casadei, R., Viroli, M., Mariani, S., Zambonelli, F.: Time-fluid field-based coordination through programmable distributed schedulers. Log. Methods Comput. Sci. **17**(4) (2021). https://doi.org/10.46298/lmcs-17(4:13)2021
45. Pianini, D., Casadei, R., Viroli, M., Natali, A.: Partitioned integration and coordination via the self-organising coordination regions pattern. Future Gener. Comput. Syst. **114**, 44–68 (2021). https://doi.org/10.1016/j.future.2020.07.032
46. Scekic, O., Schiavinotto, T., Videnov, S., Rovatsos, M., Truong, H.L., Miorandi, D., Dustdar, S.: A programming model for hybrid collaborative adaptive systems. IEEE Trans. Emerg. Top. Comput. **8**(1), 6–19 (2020). https://doi.org/10.1109/TETC.2017.2702578
47. Sudharsan, B., Yadav, P., Nguyen, D., Kafunah, J., Breslin, J.G.: Ensemble methods for collective intelligence: Combining ubiquitous ML models in iot. In: 2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, December 15-18, 2021. pp. 1960–1963. IEEE (2021). https://doi.org/10.1109/BigData52589.2021.9671901
48. Testa, L., Audrito, G., Damiani, F., Torta, G.: Aggregate processes as distributed adaptive services for the industrial internet of things. Pervasive Mob. Comput. **85**, 101658 (2022). https://doi.org/10.1016/j.pmcj.2022.101658
49. Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From distributed coordination to field calculus and aggregate computing. vol. 109 (2019). https://doi.org/10.1016/j.jlamp.2019.100486
50. Welsh, M., Mainland, G.: Programming sensor networks using abstract regions. In: Morris, R.T., Savage, S. (eds.) 1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings. pp. 29–42. USENIX (2004), http://www.usenix.org/events/nsdi04/tech/welsh.html
51. Weyns, D., Holvoet, T.: Regional synchronization for simultaneous actions in situated multi-agent systems. In: Marík, V., Müller, J.P., Pechoucek, M. (eds.) Multi-Agent Systems and Applications III, 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, Prague, Czech Republic, June 16-18, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2691, pp. 497–510. Springer (2003). https://doi.org/10.1007/3-540-45023-8_48
52. Wolf, T.D., Holvoet, T.: Designing self-organising emergent systems based on information flows and feedback-loops. In: Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, Boston, MA, USA, July 9-11, 2007. pp. 295–298. IEEE Computer Society (2007). https://doi.org/10.1109/SASO.2007.16
53. Wood, Z., Galton, A.: A taxonomy of collective phenomena. Applied Ontology **4**(3-4), 267–292 (2009). https://doi.org/10.3233/ao-2009-0071