On the Dynamic Evolution of Distributed Computational Aggregates

Giorgio Audrito
Università di Torino
Torino, Italy
giorgio.audrito@unito.it

Roberto Casadei
Università di Bologna
Cesena, Italy
roby.casadei@unibo.it

Gianluca Torta
Università di Torino
Torino, Italy
gianluca.torta@unito.it

Abstract—Engineering and programming approaches for collective adaptive systems often leverage ensemble- or group-like abstractions to characterise a subset of devices as a domain for a given task or computation. In this paper, we address the problem of programming the dynamic evolution of distributed computational aggregates, through neighbour-based coordination. This is a problem of interest, since several situated activities – especially in large-scale settings – require decentralised collaboration, and need to be sustained by limited subsets of devices. These subsets may vary dynamically due to delegation, completion of local contributions, exhaustion of resources, failure, or change in the device set induced by the openness of system boundaries.

In order to formally study and develop how distributed aggregates progressively take form by local coordination, we build on the field-based framework of aggregate processes, and extend it with techniques to support more expressive evolution dynamics. We propose novel algorithms for more effective propagation and closure of the boundaries of dynamic aggregates, based on statistics on the information speed and a notion of progressive closure through wave-like propagation. We verify the proposed techniques by simulation of a paradigmatic case study of multi-hop message delivery in mobile settings, and show increased performance and success rate with respect to previous work.

Index Terms—field-based coordination, collective adaptive systems, aggregate processes, dynamic ensembles

I. INTRODUCTION

Recent techno-scientific trends promote a vision of large-scale cyber-physical ecosystems of situated devices that self-organise to support applications like environment monitoring, Industry 4.0, and smart ecosystems. An emerging viewpoint suggests modelling and engineering such systems not just by the perspective of individual devices, but also in terms of the *overall system behaviour*. Such systems can be viewed as *Collective Adaptive Systems (CASs)* [1], [2], [3], i.e., groups of devices operating without a central coordinator and reacting to environment and input changes coherently as a whole.

Since the 2000s and earlier, research mainly stemming from the fields of coordination [4], multi-agent systems [5], self-organisation [6], [7], and swarm robotics [8] has proposed abstractions and mechanisms to engineer and program CASs. These include e.g. spatial abstractions [9], macro-programming [10], [11], field-based coordination [12], and ensemble-based approaches [7], [13], [14]. A leitmotiv in these proposals

is the definition of ways to capture *dynamic aggregates* (or ensembles), namely groups of devices that change at runtime and model providers for inputs, executors of collective tasks, recipient for multicast communications, and so on. These ensembles have proven to be crucial to promote desired collective and self-organising behaviours [7], [15], [13].

In this work, we study how dynamic aggregates could form, evolve, and cease to exist. We focus on a programming perspective, proposing algorithmic mechanisms to control their dynamics. As we consider a setting of large-scale decentralised systems engaged in ongoing computations, we assume a *"self-organisation-like"* system model of devices that compute and communicate asynchronously at discrete *rounds*, interacting with neighbours only. So, our focus is on computations that are carried out indefinitely, across multiple rounds of multiple devices, supporting e.g. monitoring, control, and self-organising activities. Thus, we adopt the *Aggregate Computing* framework [12], where such systems are programmed "as wholes" through a network-wide program manipulating *(computational) fields*, namely functions mapping device rounds to values. Specifically, we consider the notion of an *aggregate process* [16], [15], which models a transient, concurrent[1] aggregate computation running on a dynamic set of devices. Aggregate processes have been used in applications and domains like swarm-based exploration [15], peer-to-peer messaging [15], self-discovery of services in edge-cloud infrastructures [15], multi-agent plan repairing [17], and space-based coordination models [18].

The contribution is twofold. First, we provide a general formal framework, based on augmented event structures [19], [20], for modelling dynamically evolving ensembles on asynchronous networks of neighbour-interacting devices. Through this framework, we model *aggregate processes* [16] on a denotational level, without relying on the operational semantics of the field calculus [12]. Secondly, we propose novel algorithms for controlling the evolution dynamics of aggregate processes, in terms of effective propagation and shrinking (up to extinction) of such processes. We build our algorithms on *information speed* [21] statistics, a measure of space covered by data (following connectivity structures) over time. A first

---

[1]As aggregate processes are tasks on a domain of rounds, we use *concurrency* to mean that a device may run multiple processes in a same round.

algorithm exploits these statistics to guide process extinction, while a second one uses it to enact a wave-like propagation, shifting process boundaries while the process is still active.

To evaluate these techniques, we run simulations of a message delivery scenario (paradigmatic for several applications [15]) to experimentally compare them against baseline algorithms of signal-based termination through neighbourhood observation. We test the algorithms in a variety of network configurations, and quantify the improvements in terms of success rate and efficiency (i.e. number of rounds and bandwidth), showing benefits w.r.t. solutions in previous work.

The paper is organised as follows. Section II provides background on aggregate processes. Section III presents the contribution. Section IV evaluates the proposed techniques by simulation. Section V discusses related work. Section VI summarises results and outlines directions for future work.

II. BACKGROUND AND FORMAL FRAMEWORK

In this section, we first review the aggregate computing framework and its event structure model (Section II-A). Then, we introduce a formal framework to describe the dynamic formation of ensembles, applied to the notion of aggregate processes (Section II-B). This framework will also be used to present novel algorithms in Section III, and provides the benefit of not requiring the reader to go through all the details of field calculi [12]. To facilitate reading, a summary of the notation used throughout the paper is reported in Table I.

A. Aggregate Computing

Aggregate Computing [12] is a recent approach for programming CASs. It is a *macro-approach*, in the sense that the behaviour of the entire system is expressed as a single program, called an *aggregate program*, consisting of a composition of macro-level behaviours. The overall collective adaptive behaviour stems from two main ingredients: (i) an execution model for self-organisation, where each device operates at asynchronous rounds of sense-compute-interact steps; (ii) the aggregate program, specifying which data must be retrieved from sensors, processed and shared with neighbours.

1) System and execution model: The system can be modelled as a dynamic graph where nodes are *devices* and edges denote *neighbouring relationships*. Only neighbour devices can directly communicate. The neighbouring relationship may be based on logical connectivity (as in an overlay network) or physical connectivity (as in actual Wi-Fi range, so that only devices sufficiently close together can directly communicate). Indirect communication may also be possible through *stigmergy*, i.e., by perceiving and affecting the environment through *sensors* and *actuators*.

Devices operate in *rounds*. In general, each round happens asynchronously with respect to the rounds of the other devices. Each round consists of the following steps.

(i) *Context acquisition step*: the device creates a snapshot of its local context by loading its previous state, sampling sensors, and retrieving the most recent message from each of its neighbours.

(ii) *Computation step*: the device evaluates the aggregate program against its context, obtaining a result that contains (a) its local output, and (b) a coordination message (*exported data*) to be broadcast to all its neighbours.

(iii) *Interaction step*: the device broadcasts the coordination message to all its neighbours and uses the output of the computation step to drive actuators.

This is the general *aggregate execution schema*, whose details (like frequency of rounds, retention of messages from neighbours, topology management, delivery guarantees, etc.) are left to implementations and generally depend on available infrastructure and application goals.

2) Augmented event structures: An overall aggregate execution can be modelled as an *event structure* [19], where each event denotes a round. Following the approach of [20], we enrich an event structure with further information e.g. about the device in which an event happens.

Definition 1 (Augmented Event Structure): An *augmented event structure* is a 4-tuple $\mathbf{E} = \langle E, \rightsquigarrow, d, s \rangle$ where E is a countable set of *events*, $\rightsquigarrow \subseteq E \times E$ is a *messaging* relation, $d : E \to \Delta$ is a mapping from events to the devices where they happened, $s : E \to S$ is a mapping from events to (some representation of) sensors status, such that:

- for any device $\delta \in \Delta$, the set of events $E_\delta = \{\epsilon \in E \mid d(\epsilon) = \delta\}$ forms a sequence of chains, i.e., there are no distinct $\epsilon, \epsilon_1, \epsilon_2 \in E_\delta$ such that either $\epsilon \rightsquigarrow \epsilon_i$ for $i = 1, 2$ or $\epsilon_i \rightsquigarrow \epsilon$ for $i = 1, 2$;
- the transitive closure of $\rightsquigarrow$ forms an irreflexive partial order $< \subseteq E \times E$, called *causality relation*;
- the set $X_\epsilon = \{\epsilon' \in E \mid \epsilon' < \epsilon\} \cup \{\epsilon' \in E \mid \epsilon \rightsquigarrow \epsilon'\}$ is finite for all ϵ (i.e., $\rightsquigarrow$ and $<$ are locally finite).

See Figure 1 for an example of an augmented event structure. We also introduce the following concepts and notation:

- $p(\epsilon)$ denotes the previous event at the same device, i.e., the unique $\epsilon' \in E$ such that $\epsilon' \rightsquigarrow \epsilon, d(\epsilon) = d(\epsilon')$;
- $N(\epsilon)$ denotes the *neighbours* of ϵ, i.e., the set of events $\{\epsilon' \in E \mid \epsilon' \rightsquigarrow \epsilon\}$;
- $past(\epsilon)$ denotes the set of past events for ϵ, i.e., the set of events $\{\epsilon' \in E \mid \epsilon' < \epsilon\}$;
- $past_\delta(\epsilon)$ denotes the set of past events for ϵ at the same device δ, i.e., the set $\{\epsilon' \in E \mid \epsilon' < \epsilon \wedge d(\epsilon') = d(\epsilon)\}$;
- given two events $\epsilon, \epsilon' \in E$ such that $\epsilon' \rightsquigarrow \epsilon$, their *temporal distance* $lag(\epsilon, \epsilon')$ measures how much time has passed in ϵ since the interaction with ϵ' happened;
- given two events $\epsilon, \epsilon' \in E$ such that $\epsilon' \rightsquigarrow \epsilon$, their *spatial distance* $dist(\epsilon, \epsilon')$ measures how much space is covered moving from ϵ' to ϵ.

3) Computational fields: Aggregate computing is part of *field-based coordination* [22], [12], [23], an approach based on a computational interpretation of the notion of *field* as found in physics. Computational fields are distributed data structures which may be generated by the agents or the environment. Now, in the aforementioned system model, a field can be thought of as a map from devices to values or, in its "dynamic interpretation", as a map from execution rounds to values. The
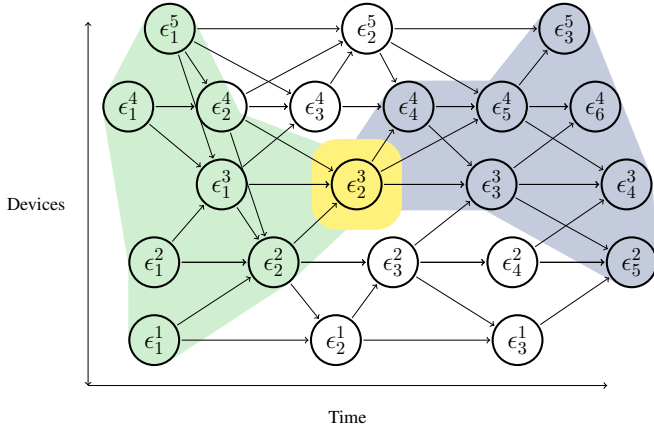
Fig. 1. Example of an event structure modelling a distributed system execution. Nodes labelled by ϵ_k^δ denote the k-th round of device δ. The yellow background highlights a reference event, from which its past (green) and future (blue) are identified through the causal relationship implied by the arrows denoting neighbour events.

latter interpretation means that a field can also be defined as a function whose domain is an event structure.

So, the aggregate programming model is based on the denotational abstraction of a *computational field* [12], [24] (*field* for short), which is essentially a function mapping each event of an event structure (*domain*) to a computational value. Fields can be used to represent collective data, results, and commands. For instance, having a set of devices query their local temperature sensor yields a field of temperature readings; or, a set of robots may coordinate their motion, which can be denoted as a field of movement vectors. Then, aggregate computing languages like FCPP [25] and ScaFi [26] provide means for describing how fields can be created, manipulated, and composed. Such languages are implementations of a minimal core language called the *field calculus* [12], which provides a formal framework characterising the semantics of field computations in terms of a minimal set of functional constructs handling (i) abstraction and composition, (ii) stateful evolution of fields, and (iii) neighbour-based communication. For the purpose of this paper, it is sufficient to note that such a formal framework represents the premier tool for implementing the algorithms and techniques described in the following sections. However, we presented them in a more general and accessible form, by referring only to the notions discussed in this section, i.e., augmented event structures and computational fields. The reader interested in the details of field calculi can refer to [12].

B. Aggregate Processes

An *aggregate process* [16], [15] is a transient, concurrent field computation that runs on a dynamic domain of devices. It is characterised by the following.

Aggregate process vs. instance, generation fields, spawning: An *aggregate process* P is a kind of field computation with given programmed behaviour. A single aggregate process 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 a process instance. New instances of an aggregate process P are spawned through a *generation field* G_P, locally producing a set of identifiers $G_P(\epsilon) = \{i \dots\}$ 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 ϵ (either being initiated by ϵ, or through propagation from previous events).

Process output and participation status: 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* $s_{P_i}(\epsilon)$ that can take the following values:

1) *internal*: the event is part of the process, and propagates it to other events of which it is a neighbour (see below);
2) *external*: the event is not part of the process;[2]
3) *border*: the event is in the process but does not propagate it to any future event (even the one on the same device).

The operation through which P_i produces its output and status is defined by its program, which we do not investigate in this paper; the interested reader may see [16] for details.

Automatic propagation of process instances to neighbours: A process instance P_i active in an event ϵ automatically propagates to any event ϵ' of which ϵ is a neighbour ($\epsilon \rightsquigarrow \epsilon'$) if and only if it returns *internal* status in ϵ. In formulas:

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

An example of evolution dynamics of two concurrent processes is provided in Figure 2. This mechanism allows devices to dynamically enter or leave the process, which can expand or shrink in space, eventually ceasing to exist when all devices quit. For instance, a device can call itself out of a process if its hop-by-hop distance (also known as *gradient* [27]) from the initiator of the process exceeds a certain threshold. Although the decision of participating or not in a process instance is ultimately local, that decision may also depend on information computed collectively within the specific process instance.

Process result vs. process status/shape computation: A process computation P_i consists of two (possibly inter-dependent) parts: the part computing the "shape" (i.e., evolving domain) of the process, and the part computing the output. These two parts can be equally important for the functionality, since a different domain of executing devices usually provides a different set of contributions, thereby affecting the overall result. Moreover, running computations on a smaller domain of devices may also provide non-functional benefits, by using fewer resources than those needed by a larger system.

Spawning aggregate process instances (construct spawn): In [16], the field calculus is extended with a construct $spawn(P, G_P)$ that runs independent instances of a field computation P, where new instances are locally generated according to a *generation field* G_P. A *spawn* expression is

---

[2]This implies discarding the output and not sharing any information generated by the execution of P_i with neighbours running the same process.
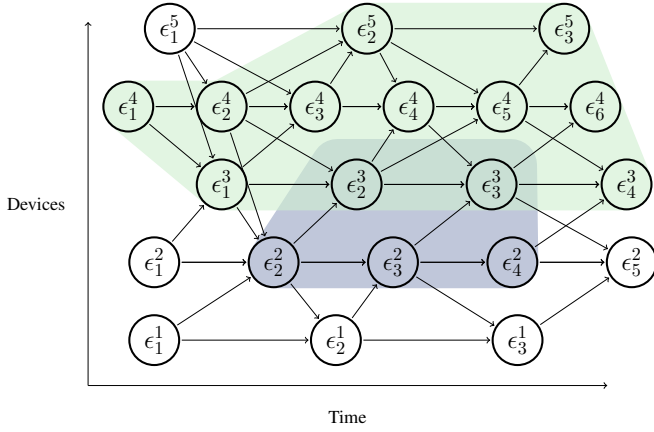
Fig. 2. Example of evolution dynamics of two concurrent aggregate process instances. Consider a process P that propagates only within 1 hop from initiator events. Instance 1 (green) is initiated by device 4, which keeps it alive until the end of the computation, by initiating it on each one if its events ϵ_1^4 to ϵ_6^4. Instance 2 (blue) is initiated only in ϵ_2^2 and ϵ_3^2, and thus starts later and closes earlier. Notice that device δ_3 in events ϵ_2^3 and ϵ_3^3 runs two process instances simultaneously. This is possible since multiple instances of the same process are allowed to overlap.

Symbol	Description
δ	Device identifier
ϵ	Event (round) identifier
ϵ_k^δ	The k-th round of device δ
$d(\epsilon)$	Device at which event ϵ occurred
$p(\epsilon)$	Previous event at the same device
$past(\epsilon)$	Set of past events of ϵ
$past_\delta(\epsilon)$	Set of past events of ϵ on device δ
$N(\epsilon)$	Neighbour past events of event ϵ
$lag(\epsilon, \epsilon')$	Temporal distance between events
$dist(\epsilon, \epsilon')$	Spatial distance between events
P	Aggregate process
P_i	Aggregate process instance, identified by i
i	Process identifier (pid)
G_P	Generator field producing pids $\{i\}$ for P
$\pi_{P_i}(\epsilon)$	Whether P_i is active in event ϵ
$O_{P_i}(\epsilon)$	Output of process instance P_i in ϵ
$s_{P_i}(\epsilon)$	Status of process instance P_i in ϵ
$s_{P_i}^*(\epsilon)$	Extended status returned by P_i in ϵ
$eSpawn_X$ $(X \in \{L, S, I, W\})$	Extended *spawn* construct (L=legacy, S=share, I=ispp, W=wispp)
$TA_X(\epsilon)$	Termination awareness of event ϵ according to *spawn$_X$* ($X \in \{L, S, I, W\}$)
$T_L(\epsilon)$	Termination predicate on event ϵ
$D^w(\epsilon)$ $(w \in \{dist, lag\})$	Shortest-path distances based on weight function $w(\epsilon, \epsilon')$ and source predicate $src(\epsilon)$
$Sl(\epsilon)$	Slowness predicate on event ϵ
θ	Minimum information speed allowed

evaluated round by round, and in different rounds the field G may vary. In a round ϵ in which, e.g., $i \in G_P(\epsilon)$, a new instance of P with identifier i will be spawned locally. We remark that the computations of different instances P_i and P_j are fully independent and do not share any data among them. Hence, they represent separate activities, each with its peculiar evolution and history.

Output of a spawn expression: The output of a $spawn(P, G_P)$ expression in an event ϵ is the set of pairs $\{(i, O_{P_i}(\epsilon)), \ldots\}$ for which $\pi_{P_i}(\epsilon)$ is true and $s_{P_i}(\epsilon)$ is not *external*. Notice that globally this is a *field* of maps, while locally to an individual device it is a (possibly empty) map. In this work, we enhance this construct and propose algorithmic techniques to improve propagation and shrinking dynamics.

C. Example: situated service discovery

Consider a network of devices in a smart city that may offer and request services (e.g., computing services as in volunteer computing [28]). Interactions are limited to neighbours e.g. for scalability, latency, or privacy reasons. Service requests scan the surroundings of the requester device for offers including cost and service-level agreements (SLAs), so that the requester can choose and consume the offer that it deems best. The requester and service provider may be hops away, thus other devices need to act as relays for data and results. This logic can be expressed through a $spawn(P, G_P)$ where G_P is the set of service requests to be spawned, and P is an aggregate computation that e.g. spreads the process in space until a certain distance threshold is covered, through a gradient [21], and collects offers [29] by aggregating them while descending the spanning tree induced by the gradient field.

III. TECHNIQUES FOR DYNAMIC ENSEMBLES

In this section, we describe novel techniques for dynamic ensemble formation by means of extensions to the basic *spawn* function covered in Section II-B. The *extended spawn eSpawn* takes a function P similarly to *spawn*, but P can return an *extended status* that can take the additional value *terminated*. Such a value is intended to be interpreted by *eSpawn* as an indication by P that the process should terminate not only in the current node, but the termination should spread to all the nodes in the network. In the following subsections we shall propose four alternative versions of *eSpawn* that aim to achieve the best performance in terms of minimizing the (computational and networking) resources required to both guide process propagation, and handle process termination. As we shall see, those versions will re-interpret the states returned by P (*terminated* as well as the others), providing means to control process propagation and termination.

A. Baselines

A first version of *eSpawn* has already been presented in the literature, although with a different name [16], [15]. In this section, we reformulate it in a way homogeneous to the other versions we have devised, to facilitate comparisons and its use as a baseline in experiments (Section IV). Moreover,

we propose a first alternative that, although simple and natural to realise, leads to significant performance improvements.

First, we consider the *legacy* version of *eSpawn* found in literature [16], which we denote as $eSpawn_L$, and describe it in terms of how it determines the process status at each event. When a process instance P_i is active in an event ϵ (i.e., $\pi_{P_i}(\epsilon)$ is *true*), an extended status $s^*_{P_i}(\epsilon)$ is returned by P, and it is interpreted into a "regular" status $s_{P_i}(\epsilon)$ by *eSpawn*. Towards this aim, we define *termination awareness* for a process instance P_i and event ϵ as follows.

Definition 2: Predicate $TA_L(\epsilon)$ (termination awareness) denotes the fact that event ϵ is termination-aware, i.e., either $s^*_{P_i}$ demands termination in ϵ, or some neighbour event was termination-aware since its previous round. In formulas:

$$TA_L(\epsilon) := s^*_{P_i}(\epsilon) = terminated \text{ or } \exists \epsilon' \in N'(\epsilon). \ TA_L(p(\epsilon')). \tag{1}$$

Above, $N'(\epsilon) = N(\epsilon) - p(\epsilon) \cup \{\epsilon\}$, that is the replacement of $p(\epsilon)$ with ϵ in $N(\epsilon)$.

According to this definition, if $TA_L(\epsilon)$ becomes *true*, it will not go back to *false* in future events ϵ' on device $d(\epsilon)$ until $\pi_{P_i}(\epsilon')$ becomes *false*. This has the purpose of keeping track in device $d(\epsilon)$, for some rounds, that P should terminate.

Actual termination occurs when both ϵ and all its neighbours $N(\epsilon)$ agree that the process should terminate, as computed by the termination predicate $T_L(\epsilon)$ defined by:

$$T_L(\epsilon) := \forall \epsilon' \in N'(\epsilon). \ TA_L(\epsilon'). \tag{2}$$

When $T_L(\epsilon)$ becomes *true*, it is time for the process to terminate at device $d(\epsilon)$, with $s_{P_i}(\epsilon)$ taking value *external* (regardless of the value $s^*_{P_i}(\epsilon)$ returned by P). If this is not the case, but $s^*_{P_i}(\epsilon) = terminated$, then $s_{P_i}(\epsilon) = internal$, indicating that ϵ is still within P_i (in order to propagate termination). In all other cases, $s_{P_i}(\epsilon) = s^*_{P_i}(\epsilon)$.

As mentioned above, a significant improvement can be easily obtained by exploiting the semantics of the *share* operator recently introduced in Field Calculus (FC) [30], instead of the *rep* and *nbr* operators used in $eSpawn_L$. In terms of computation on the event structure, the $eSpawn_S$ extension (where subscript S is for *share*) allows a more efficient definition of predicate $TA_S(\epsilon)$ compared to Equation (1), by directly accessing the predicate in neighbour events:

$$TA_S(\epsilon) := s^*_{P_i}(\epsilon) = terminated \text{ or } \exists \epsilon' \in N(\epsilon). \ TA_S(\epsilon'). \tag{3}$$

The propagation of termination awareness TA_S is clearly faster in this case, since event ϵ directly exploits the values of TA_S of its neighbours ϵ', instead of that of their predecessors $p(\epsilon')$. Actual termination $T_S(\epsilon)$ in $eSpawn_S$ is defined as in Equation (2), but based on the TA_S defined by Equation (3).

B. Exploiting Information Speed

We now describe a further extensions to *spawn*, addressing some shortcomings of $eSpawn_S$: in particular, the possible resurgence of terminated processes due to some isolated nodes not receiving the termination signal. The idea on which it is based, inspired by the BIS algorithm [21], is that of estimating the spatial and temporal distances of each event ϵ in the process from the source event ϵ_0 that has started the process. In case these two distances correspond to an *information speed* that is below a certain threshold (more details below), it is taken as an indication of a likely disconnection from the source, prompting the device $d(\epsilon)$ to leave the process.

We call this extension $eSpawn_I$ (where subscript I is for *Information Speed-based Process Propagation*, also shortened as *ISPP*). In $eSpawn_I$, termination awareness TA_I computed as TA_S Equation (3). Differently from $eSpawn_S$, though, TA_I becoming true implies a transformation in the returned status, so that $s^*_{P_i}(\epsilon) = internal$ gets converted to $s_{P_i}(\epsilon) = border$ (cf. Equation (6)). This implies that termination-aware nodes *do not propagate the process* to their neighbours, slowing down the process expansion and helping termination to catch up with it. Furthermore, there is no need for a termination predicate T_I: since termination-aware nodes do not propagate the process (not even to their next event), once every neighbour is termination-aware the process naturally stops anyway.

In addition to this difference, $eSpawn_I$ also features another term called $Sl(\epsilon)$ to detect a *slow* information propagation (hence likely disconnection from the source). Let us denote with $D^w(\epsilon)$ a classic shortest-path distance function based on weights $w(\epsilon, \epsilon')$ (where $\epsilon' \rightsquigarrow \epsilon$) and a *source* predicate $src(\epsilon)$. In each event, this distance estimate is updated to the smallest distance through a neighbour event, as in a step of the Bellman-Ford algorithm (but on an event structure):

$$D^w(\epsilon) := \begin{cases} 0 & \text{if } src(\epsilon) \\ \min\{D^w(\epsilon') + w(\epsilon, \epsilon') : \ \epsilon' \in N(\epsilon)\} & \text{otherwise.} \end{cases} \tag{4}$$

Through D^w, an estimation of the spatial or temporal distance of events from sources can be obtained after a few rounds of computation, even for those far from sources. Assume that $src(\epsilon)$ is true in all events on the device that spawned the process by providing pid i in the generation field G_P, and w is either the spatial distance $dist$ or the temporal distance lag (cf. Section II-A2).

An important property of D^{dist} is that, since it is a minimisation and $dist(\epsilon, p(\epsilon))$ is always 0, it can never increase from two events $p(\epsilon)$ and ϵ on the same device $d(\epsilon)$: indeed, even if the values from other neighbours increase, $D^{dist}(\epsilon)$ can keep the previous value $D^{dist}(p(\epsilon))$. On the contrary, in D^{lag}, the weight $lag(\epsilon, p(\epsilon))$ is the time interval between two rounds, which is positive hence $D^{lag}(\epsilon)$ can increase between two successive events on the same device. In fact, it can be shown that $D^{lag}(\epsilon)$ is always equal to the temporal distance between the current event ϵ and the most recent source event in ϵ's past (modulo imprecisions in lag measurements). Thus, we define *slowness* as follows.

Definition 3: Predicate $Sl(\epsilon)$ (slowness) denotes the fact that the information speed detected at event ϵ is *too slow*, i.e.:

$$Sl(\epsilon) := D^{dist}(\epsilon) \leq \theta(D^{lag}(\epsilon) - \Delta_t) \tag{5}$$

where Δ_t is the average time interval between rounds, and θ is a constant representing the minimum speed of information

that we are willing to allow (cf. Section III-D).

When $Sl(\epsilon)$ becomes true, it causes an event to also enter the *border* state. Summarising, the status s_{P_i} is computed as:

$$s_{P_i}(\epsilon) := \begin{cases} external & \text{if } s_{P_i}^*(\epsilon) = external, \text{ otherwise} \\ border & \text{if } s_{P_i}^*(\epsilon) = border \text{ or } TA_I(\epsilon) \text{ or } Sl(\epsilon) \\ internal & \text{otherwise.} \end{cases}$$
(6)

An important role of Sl in making a process terminate everywhere in the network is when devices move, hence possibly quitting the main part of the process (the one originated from its source). In such a case, events ϵ happening in the connected devices are still *not* slow, because newer events happening in the source continue to spread a 0 temporal distance that leads to $D^{lag}(\epsilon)$ to be (with high likelihood) too low for Equation (5). On the other hand, events happening on the disconnected devices no longer receive information from the source, so that their D^{lag} keeps increasing while their estimated spatial distance D^{dist} stays constant, up to a point when $Sl(\epsilon)$ becomes $true$, making them enter the border state, stopping process propagation and ultimately terminating that disconnected part. Without the Sl mechanism, isolated groups of devices would continue to run the process, and, even worse, would propagate it to any devices they dynamically connect to through movements of the devices, possibly resurrecting it after the process has already terminated its task elsewhere.

C. Wave-like Propagation

The last extension of $spawn$ we introduce in this paper is $eSpawn_W$ (where subscript W is for *Wave-like ISPP*). Technically, this version is similar to $eSpawn_I$, but the two versions exhibit fundamentally different behaviours: while $eSpawn_I$ aims at terminating isolated instances of the process that would not become aware in due time that the process must terminate everywhere, $eSpawn_W$ acts during a phase when the process is still propagating (e.g., the message has not been delivered yet to destination). Thus, it aims at modifying the dynamic evolution of alive processes, by removing them from selected devices that have already acted as propagators, and are no longer required to support the process.

Specifically, the difference between $eSpawn_W$ and $eSpawn_I$ resides in the definition of the src predicate used by D_{dist} and D_{lag} to determine whether an event ϵ is a source event. In $eSpawn_W$, $src(\epsilon)$ is only true in the very first event when the process is generated by providing pid i in the generation field G_P, and *not* in the following events on the same device (unlike $eSpawn_I$). This apparently minor difference has large consequences in the algorithm behaviour: as the source ephemerally disappears, every event behave as if being in a disconnected part, eventually becoming *slow* and leaving the process (including the original source itself). This leads to a wave of termination that starts from the source device and propagates outwards, with some time lag after the propagation of the process itself. At any given time, the process is thus active only on a set of events with similar spatial distances from ϵ_0, leading to a wave-like propagation.

This still allows the process to travel far through the network, while keeping its spatial extension low at all times.

D. Adapting to Different Scenarios

The newly introduced extensions both rely on a crucial parameter θ, the minimum information speed that we are willing to allow. Correctly tuning this parameter is crucial for obtaining the best performance. If θ becomes higher than the average information speed in the network, processes terminate prematurely, failing to accomplish their tasks. In order to avoid this scenario, the estimate of θ needs to be on the conservative side: however, if it is too low, the behaviour of the new extensions will degenerate to be very similar to $eSpawn_S$.

Furthermore, a one-fits-all number for θ is impossible, as the information speed depends on many parameters: time intervals between rounds t and their variance $tvar$, communication radius r, dimensionality of the space n, device density $dens$, movement speed $speed$ and whether propagation is allowed through a single path or multiple paths. If the process is restricted to a single path, the theoretical-based estimation in [21] of *single-path* information speed can be used:

$$\theta_{sp} = \frac{8n(1 + tvar^2)r}{4(n+1)t} + \frac{speed}{2}.$$
(7)

For instance, this would apply to the tree topology scenario in the experiments. If instead the propagation is allowed through multiple paths, a theoretical-based approach is more complicated, and empirical estimates (obtained through simulation tools such as FCPP [25]) are most useful. In the particular case where $n = 2$, we found that the following formula behaved reasonably well in a large number of settings:

$$\theta_{mp} = (0.08 dens - 0.7) speed + (0.075 dens^2 - 1.6 dens + 11) r/t$$
(8)

IV. EXPERIMENTS

We have executed a number of experiments on the proposed algorithms using the FCPP simulator [25]. The basic use case implemented by all experiments is a network of devices where, at some point in time, a source device δ_F (*from*) sends a message through a process to reach a destination device δ_T (*to*). It is worth noting that, in the simulation, there is a global clock (unlike in a real world scenario), which however is not available to the program run by individual nodes. Round durations are not identical (see parameter *tvar* below), and the simulator provides to each node the exact spatial distance $dist$ and temporal distance lag between neighbour events, needed for computing the speed of information in $eSpawn_I$ and $eSpawn_W$ (see Section III-B). In a real scenario such measures would be estimates subject to errors.

We distinguish between two scenarios based on the topology followed by communications. In the $spherical$ topology, messages originate in the δ_F device and spread radially in 3D trying to reach the δ_T device. In the $tree$ topology, the devices are organised in a spanning tree, and communications follow the edges of the tree from δ_F to δ_T. In both scenarios, the events happening at the δ_T device make the process function
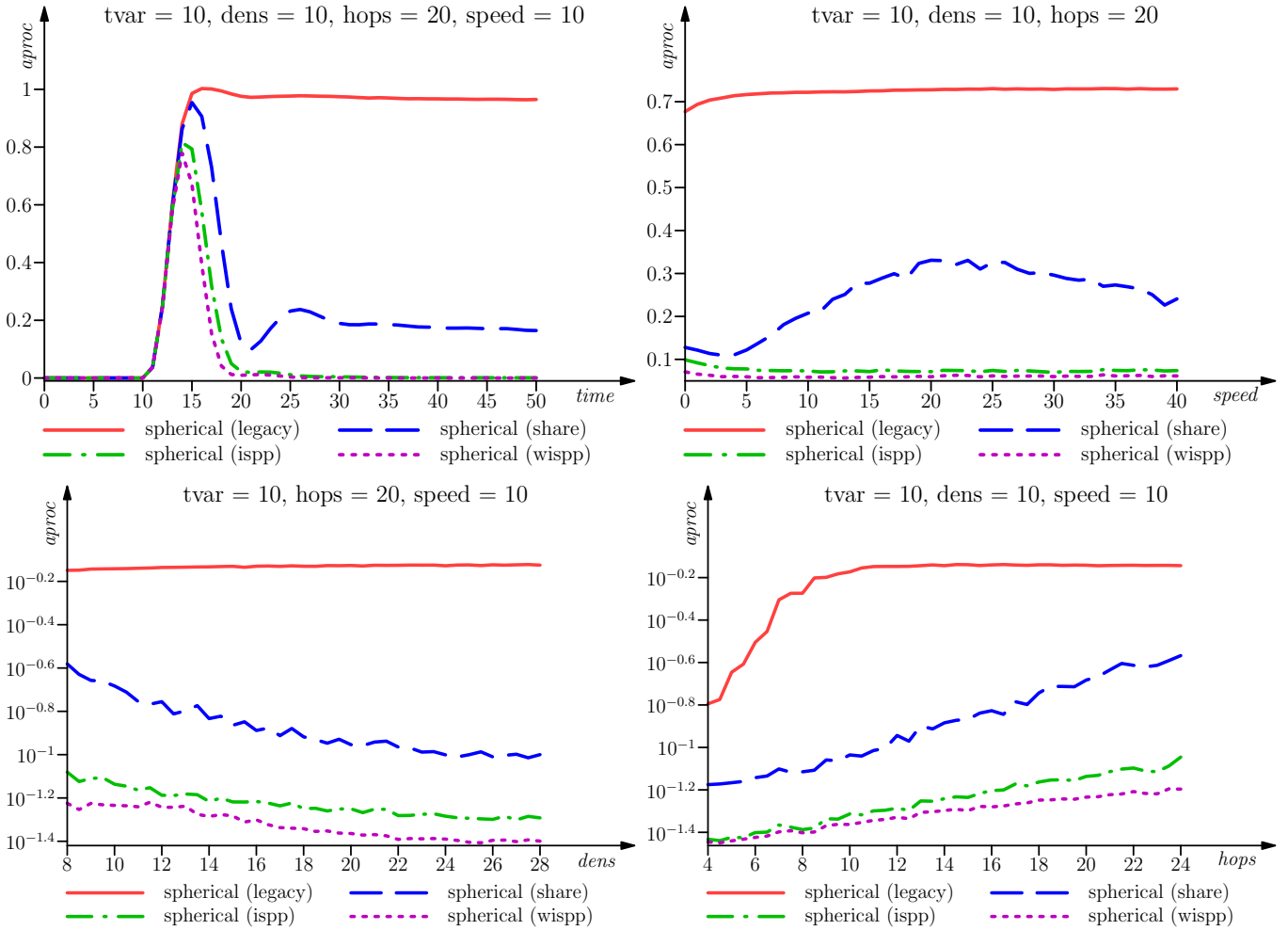
Fig. 3. Average processes with spherical topology varying time, *speed*, *dens*, *hops*.

P return status *terminated*, since the reception of the message implies the termination of the process in the whole network. For simplicity, in each run we generate only one process at time $t = 10$ and wait until time $t = 50$ for its completion. We also verified that results do not change by generating several processes in each test, as expected since processes are independent of one another. We considered 80.000 total scenarios, with 200 different random seeds and 40 different values of four parameters:

1) *tvar* (relative variance of round durations as percentage), varying from 0% (quasi-synchronous rounds) to 40% (highly asynchronous rounds), in steps of 1%;
2) *dens* (average number of neighbour devices for a device), varying from 8 (very sparse network) to 28 (very dense network), in steps of 0.5;
3) *hops* (average diameter of the network in hops), varying from 4 (small networks) to 24 (relatively large networks), in steps of 0.5;
4) *speed* (movement speed as percentage of communication radius over round duration). varying from 0% (static nodes) to 40% (highly mobile nodes), in steps of 5%.

The repository of experiments is publicly available online for reproducibility.[3]

A. Spherical Topology

We first discuss results with the spherical topology. Figure 3 shows the average number of active processes (*aproc*) for each version of *eSpawn*: $eSpawn_L$ (*legacy*), $eSpawn_S$ (*share*), $eSpawn_I$ (*ispp*), and $eSpawn_W$ (*wispp*). The top left graph shows the variation of *aproc* over 50 time instants, averaging over 200 random cases. The other graphs show the variation of *aproc* over, respectively, *speed*, *dens*, and *hops*; with each point averaged over 50 time instants of 200 random cases.

In resource consumption, $eSpawn_L$ is much worse than its extensions, while $eSpawn_I$ and $eSpawn_W$ still outperform $eSpawn_S$. All the algorithms successfully deliver the message at about the same time (not shown in the figure). These results are confirmed varying *speed*, *dens*, and *hops*.

In Figure 3 (bottom left) it is also visible the small relative difference of performance between $eSpawn_W$ and $eSpawn_I$ (with the former outperforming the latter), when varying the

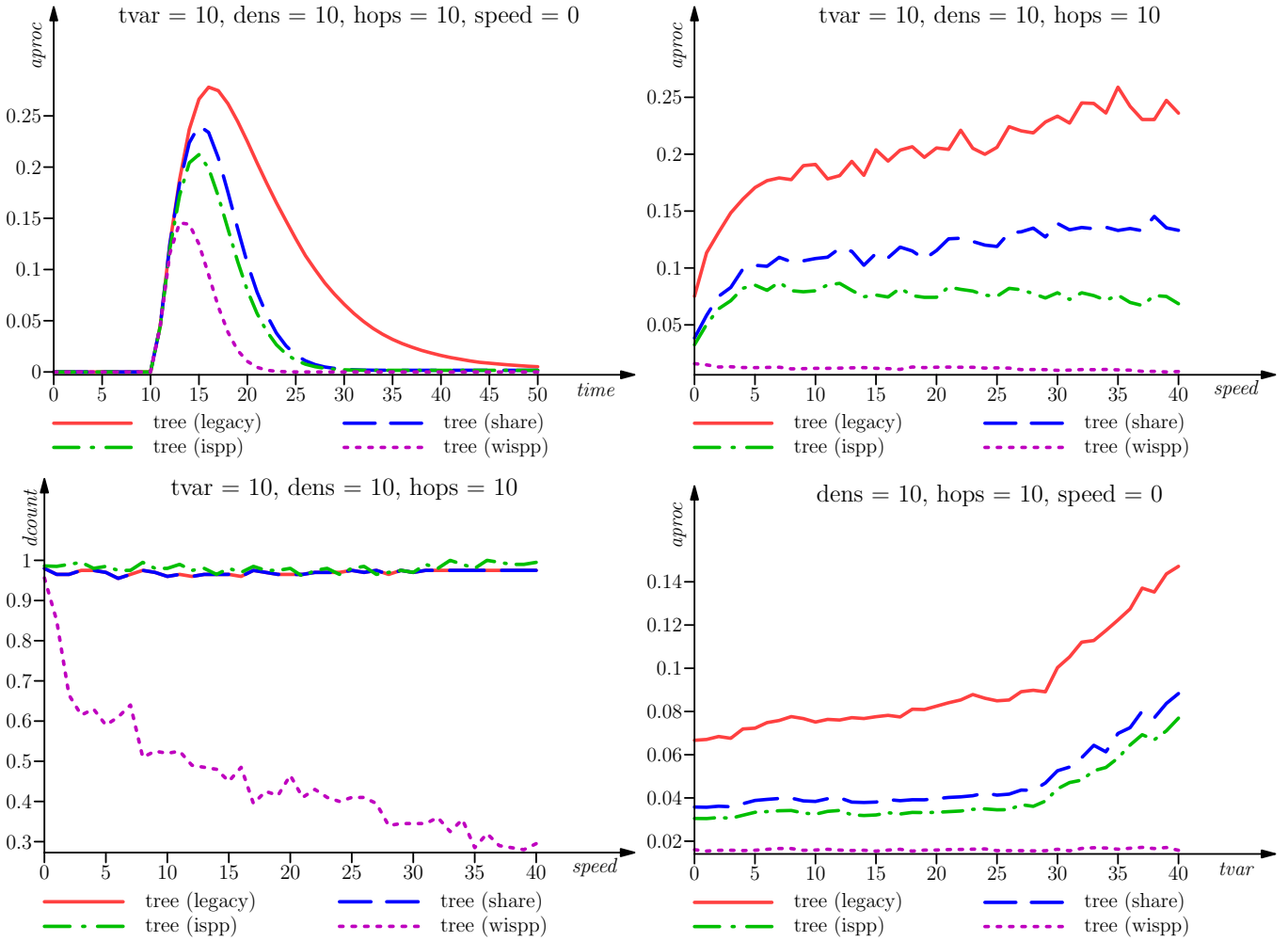[3]https://github.com/fcpp-experiments/process-management

Fig. 4. Average processes with tree topology varying time, *speed* (with delivered messages count), *tvar*.

density of devices, while keeping the other parameters fixed. Additionally, when we vary the "hop" size of the network (bottom right), we detect an increasing gap between the performance of $eSpawn_W$ and $eSpawn_I$ as the number of hops increases. This is expected, as the wave of terminating nodes in $eSpawn_W$ has more time to be effective in large-diameter networks. Finally, we have also run tests by varying *tvar* while keeping the other parameters fixed. We do not report the resulting plot since it does not show particular trends, beside confirming the relative performance order between the four versions of $eSpawn$.

B. Tree Topology

For the *tree* scenario, we implemented an adaptive algorithm computing the spanning tree based on [31]. Then, we guide process propagation by first following tree parents towards the root, and then following children that include the destination in their routing table (also computed through a literature adaptive algorithm, called *single-path collection* [32]).

Figure 4 (top left) shows the average number of active processes over time for each version of $eSpawn$ (parameters indicated in the plots). As expected, the percentage of nodes running the process at any given time is significantly lower than for the spherical topology (cf. Figure 3). The $eSpawn_L$ is worse than its extensions, while $eSpawn_W$ clearly outperforms both $eSpawn_S$ and (to a lesser degree) $eSpawn_I$. All the algorithms successfully deliver the message at about the same time (not shown in the figure).

As the tree structure is continuously adapted, it is expected that when devices move (i.e., their speed is not 0) this could impact the performance of the algorithms. Figure 4 (top right and bottom left) shows both the average percentage of process nodes and the delivery count while varying the device speed and keeping the other parameters fixed. For this parameter setting, all the algorithms except $eSpawn_W$ deliver the message even at high speeds.

A similar result is visible in Figure 4 (bottom right) where we vary the *tvar* of the rounds of devices, while keeping the other parameters fixed. Again, all the algorithms except $eSpawn_W$ deliver the message even at high variance, while $eSpawn_W$ performance degrades, although more smoothly than in the case of speed increase.

Similarly to the spherical topology, varying the density of the devices, while keeping the other parameters fixed, just makes more visible the relative difference of performance between $eSpawn_W$ and $eSpawn_I$. Similar outcomes are obtained by varying the hops, and are thus not presented here.

V. Related Work

In the following, we distinguish multiple clusters of works that share some commonalities with our approach to distributed computational aggregate formation.

Pattern languages: The proposed work can be framed within the field of self-organising multi-robot pattern formation [33]. The survey [9] on spatial computing identifies *pattern languages* as a class of works aiming to produce spatial, geometrical, or topological patterns in amorphous computers made of a several simple, unreliable devices locally communicating with one another. For instance, Origami Shape Language (OSL) [34], enables to build shapes on a surface through a sequence of flat folding operations. An OSL program is then implemented by uniform cell programs leveraging gradients, neighbourhood queries, and other local operations like local folding. Another example is Growing Point Language [35], which uses trajectories of "growing points" (mobile computations) diffusing across nodes to form patterns. However, unlike aggregate processes, growing points are active at a single domain at a time. These works are related as they propose mechanisms for building shapes incrementally in systems of neighbour-interacting devices. However, they tend to focus on the shape of groups and neglect the information processing carried out in the defined domains.

Ensemble-based approaches: An *ensemble* is a dynamic group of devices that forms to support group-level tasks. In *Distributed Emergent Ensembles of Components (DEECo)* [13], ensembles are characterised by a membership condition that expresses how a set of components get bound together. Within an ensemble, the components interact by implicit knowledge exchange. Aggregate processes can also be seen as regulated through a membership condition, i.e., the status determining whether the node is willing to participate in the process; however, ensemble formation is a dynamic activity that runs on a given communication topology that also regulates interactions within processes. *Service Component Ensemble Language (SCEL)* [36] is a language that enables to express the behaviour of ensembles interacting via attributed-based communication. Ensemble formation is thus regulated through predicates over attributes exposed by components. This is different from aggregate processes, where communication is constrained by both the given neighbouring relationship and process membership. In summary, both in DEECo and SCEL, the key aspect of ensemble domain propagation and shrinking addressed in this paper is not directly captured.

Clustering and Area Formation: Swarm clustering [37] brings the data clustering problem into swarm settings, where the idea is to group agents into *clusters* such that the agents in the same cluster are more correlated to each other (e.g., spatially or temporally) than to the agents belonging to other clus-ters. For instance, in [38], a mathematical model for cluster-based group formation is proposed that takes inspiration from bee foraging and recruitment in order to assemble groups with complementary skills. A similar problem involves organising a system into regions or areas to solve a certain problem with a configurable level of decentralisation [39], cf. the *Self-organising Coordination Regions* pattern [40]. Swarm clusters and such pattern can, indeed, be expressed with aggregate processes. Also, aggregate processes seamlessly model the case where clusters need to overlap, which may be instrumental for conflict resolution or inter-regional coordination. Vice versa, clustering processes could be used to regulate the formation of aggregate process domains; however, these could not naturally cover all the possible evolution dynamics that CASs may exhibit (e.g., wave-like ones).

Spreading and epidemic processes over networks: The topic of this paper is also potentially related to spreading and epidemic processes in time-varying and complex networks [41], [42]. Among the key distinguishing factors between those works and this one there are the system model (cf. Section II) and the emphasis on programmability of the logic for incremental process domain evolution. However, studying the dynamics of aggregate processes via tools and methods from network science could be an interesting future work.

VI. Conclusion

In this paper, we have addressed the dynamic evolution of distributed computational ensembles (or aggregates), using descriptions over augmented event structures. Starting from the field-based framework of aggregate processes [16], [18], [15], we have proposed algorithms for effective propagation and closure of group-wise processes, providing trade-offs in terms of efficiency (e.g., in terms of rounds and messages), design simplicity, and functionality. Specifically, we leveraged information speed statistics to propose shrinking modalities and wave-like propagation strategies. Then, we have shown experimentally that the novel algorithms improve over a baseline given by previous work in [16], [15].

We believe that the algorithms proposed in this work can provide a benefit to existing scenarios and coordination models built on aggregate processes, such as *situated tuples* [18] and *Self-organising Coordination Regions* [40]. In future work, we consider studying aggregate processes with methods found in areas like complex networks and epidemics (briefly reviewed in Section V), e.g., to devise formal guarantees on aspects of evolution dynamics. Finally, we would also consider exploiting the ability of aggregate processes of dynamically forming cohesive groups of devices to promote *self-improving system integration (SISSY)* goals [43].

References

[1] R. D. Nicola, S. Jähnichen, and M. Wirsing, "Rigorous engineering of collective adaptive systems: special section," *Int. J. Softw. Tools Technol. Transf.*, vol. 22, no. 4, pp. 389–397, 2020.

[2] A. Bucchiarone, M. D'Angelo, D. Pianini, G. Cabri, M. De Sanctis, M. Viroli, R. Casadei, and S. Dobson, "On the social implications of collective adaptive systems," *IEEE Technol. Soc. Mag.*, vol. 39, no. 3, pp. 36–46, 2020.

[3] A. Farahani, G. Cabri, and E. Nazemi, "Self-* properties in collective adaptive systems," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp Adjunct 2016.* ACM, 2016, pp. 1309–1314.

[4] T. W. Malone and K. Crowston, "The interdisciplinary study of coordination," *ACM Comput. Surv.*, vol. 26, no. 1, pp. 87–119, 1994.

[5] J. Ferber, *Multi-agent systems - an introduction to distributed artificial intelligence.* Addison-Wesley-Longman, 1999.

[6] D. Ye, M. Zhang, and A. V. Vasilakos, "A survey of self-organization mechanisms in multiagent systems," *IEEE Trans. Syst. Man Cybern. Syst.*, vol. 47, no. 3, pp. 441–461, 2017.

[7] S. von Mammen, S. Tomforde, and J. Hähner, "An organic computing approach to self-organizing robot ensembles," *Frontiers Robotics AI*, vol. 3, p. 67, 2016.

[8] M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo, "Swarm robotics: a review from the swarm engineering perspective," *Swarm Intell.*, vol. 7, no. 1, pp. 1–41, 2013.

[9] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll, "Organizing the aggregate: Languages for spatial computing," in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments.* IGI Global, 2013, ch. 16, pp. 436–501.

[10] R. Newton and M. Welsh, "Region streams: Functional macroprogramming for sensor networks," in *Workshop on Data Management for Sensor Networks*, 2004, pp. 78–87.

[11] R. Casadei, "Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling," *CoRR*, vol. abs/2201.03473, 2022. [Online]. Available: https://arxiv.org/abs/2201.03473

[12] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini, "From distributed coordination to field calculus and aggregate computing," *J. Log. Algebraic Methods Program.*, vol. 109, 2019.

[13] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, "DEECO: an ensemble-based component system," in *CBSE'13, Proceedings of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering.* ACM, 2013, pp. 81–90.

[14] R. Hennicker and A. Klarl, "Foundations for ensemble modeling - the helena approach - handling massively distributed systems with elaborate ensemble architectures," in *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi*, ser. LNCS, vol. 8373. Springer, 2014, pp. 359–381.

[15] R. Casadei, M. Viroli, G. Audrito, D. Pianini, and F. Damiani, "Engineering collective intelligence at the edge with aggregate processes," *Eng. Appl. Artif. Intell.*, vol. 97, p. 104081, 2021.

[16] ——, "Aggregate processes in field calculus," in *International Conference on Coordination Languages and Models.* Springer, 2019, pp. 200–217.

[17] G. Audrito, R. Casadei, and G. Torta, "Fostering resilient execution of multi-agent plans through self-organisation," in *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2021, Companion Volume.* IEEE, 2021, pp. 81–86.

[18] R. Casadei, M. Viroli, A. Ricci, and G. Audrito, "Tuple-based coordination in large-scale situated systems," in *Coordination Models and Languages - 23rd International Conference, COORDINATION 2021, Proceedings*, ser. LNCS, vol. 12717. Springer, 2021, pp. 149–167.

[19] M. Nielsen, G. D. Plotkin, and G. Winskel, "Petri nets, event structures and domains, part I," *Theor. Comput. Sci.*, vol. 13, pp. 85–108, 1981.

[20] G. Audrito, J. Beal, F. Damiani, and M. Viroli, "Space-time universality of field calculus," in *Coordination Models and Languages*, ser. LNCS, vol. 10852. Springer, 2018, pp. 1–20.

[21] G. Audrito, F. Damiani, and M. Viroli, "Optimal single-path information propagation in gradient-based algorithms," *Sci. Comput. Program.*, vol. 166, pp. 146–166, 2018.

[22] M. Mamei and F. Zambonelli, "Field-based coordination for pervasive computing applications," in *Bio-Inspired Computing and Communication, 1st Workshop on Bio-Inspired Design of Networks, BIOWIRE 2007*, ser. LNCS, vol. 5151. Springer, 2007, pp. 376–386.

[23] D. Xiao and R. J. Hubbold, "Navigation guided by artificial force fields," in *Proceeding of the CHI '98 Conference on Human Factors in Computing Systems.* ACM, 1998, pp. 179–186.

[24] M. Mamei, F. Zambonelli, and L. Leonardi, "Co-fields: Towards a unifying approach to the engineering of swarm intelligent systems," in *Engineering Societies in the Agents World III, 3rd International Workshop, ESAW 2002, Revised Papers*, ser. LNCS, vol. 2577. Springer, 2002, pp. 68–81.

[25] 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.

[26] R. Casadei, M. Viroli, G. Audrito, and F. Damiani, "FScaFi: A core calculus for collective adaptive systems programming," in *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles*, ser. LNCS, vol. 12477. Springer, 2020, pp. 344–360.

[27] G. Audrito, R. Casadei, F. Damiani, and M. Viroli, "Compositional blocks for optimal self-healing gradients," in *11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017.* IEEE Computer Society, 2017, pp. 91–100.

[28] T. M. Mengistu and D. Che, "Survey and taxonomy of volunteer computing," *ACM Comput. Surv.*, vol. 52, no. 3, pp. 59:1–59:35, 2019. [Online]. Available: https://doi.org/10.1145/3320073

[29] G. Audrito, R. Casadei, F. Damiani, D. Pianini, and M. Viroli, "Optimal resilient distributed data collection in mobile edge environments," *Comput. Electr. Eng.*, vol. 96, no. Part, p. 107580, 2021. [Online]. Available: https://doi.org/10.1016/j.compeleceng.2021.107580

[30] G. Audrito, J. Beal, F. Damiani, D. Pianini, and M. Viroli, "Field-based coordination with the share operator," *Log. Methods Comput. Sci.*, vol. 16, no. 4, 2020.

[31] J. Beal, "Flexible self-healing gradients," in *Proceedings of the 2009 ACM symposium on Applied Computing*, 2009, pp. 1197–1201.

[32] M. Viroli, G. Audrito, J. Beal, F. Damiani, and D. Pianini, "Engineering resilient collective adaptive systems by self-stabilisation," *ACM Trans. Model. Comput. Simul.*, vol. 28, no. 2, pp. 16:1–16:28, 2018.

[33] H. Oh, A. R. Shirazi, C. Sun, and Y. Jin, "Bio-inspired self-organising multi-robot pattern formation: A review," *Robotics Auton. Syst.*, vol. 91, pp. 83–100, 2017.

[34] R. Nagpal, "Programmable self-assembly: constructing global shape using biologically-inspired local interactions and origami mathematics," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2001. [Online]. Available: http://hdl.handle.net/1721.1/86667

[35] D. Coore, "Botanical computing: a developmental approach to generating interconnect topologies on an amorphous computer," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999. [Online]. Available: http://hdl.handle.net/1721.1/80483

[36] R. D. Nicola, M. Loreti, R. Pugliese, and F. Tiezzi, "A formal approach to autonomic systems programming: The SCEL language," *ACM Trans. Auton. Adapt. Syst.*, vol. 9, no. 2, pp. 7:1–7:29, 2014.

[37] C. Lee, M. Kim, and S. Kazadi, "Robot clustering," in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, 2005.* IEEE, 2005, pp. 1449–1454.

[38] D. S. dos Santos and A. L. C. Bazzan, "Distributed clustering for group formation and task allocation in multiagent systems: A swarm intelligence approach," *Appl. Soft Comput.*, vol. 12, no. 8, pp. 2123–2131, 2012.

[39] D. Weyns and T. Holvoet, "Regional synchronization for simultaneous actions in situated multi-agent systems," in *Multi-Agent Systems and Applications III, 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, Proceedings*, ser. LNCS, vol. 2691. Springer, 2003, pp. 497–510.

[40] R. Casadei, D. Pianini, M. Viroli, and A. Natali, "Self-organising coordination regions: A pattern for edge computing," in *Coordination Models and Languages - 21st International Conference, COORDINATION 2019 Proceedings*, ser. LNCS, vol. 11533. Springer, 2019, pp. 182–199.

[41] C. Nowzari, V. M. Preciado, and G. J. Pappas, "Analysis and control of epidemics: A survey of spreading processes on complex networks," *IEEE Control Systems Magazine*, vol. 36, no. 1, pp. 26–46, 2016.

[42] M. Cremonini and S. Maghool, "The dynamical formation of ephemeral groups on networks and their effects on epidemics spreading," *Scientific Reports*, vol. 12, no. 1, pp. 1–10, 2022.

[43] K. L. Bellman, J. Botev, A. Diaconescu, L. Esterle, C. Gruhl, C. Landauer, P. R. Lewis, P. R. Nelson, E. Pournaras, A. Stein, and S. Tomforde, "Self-improving system integration: Mastering continuous change," *Future Gener. Comput. Syst.*, vol. 117, pp. 29–46, 2021.