Complex & Intelligent Systems (2026) 12:63
https://doi.org/10.1007/s40747-025-02141-y

ORIGINAL ARTICLE

FieldVMC: an asynchronous model and platform for self-organising morphogenesis of artificial structures

Angela Cortecchia[1] · **Giovanni Ciatto**[1] · **Roberto Casadei**[1] · **Danilo Pianini**[1]

Received: 7 June 2025 / Accepted: 9 October 2025

Abstract

The vascular morphogenesis controller (VMC) is an approach to structure development inspired by the way plants branch and distribute nutrients. It has proven useful to guide shape formation in modular robotics as well as resource distribution in hierarchically-structured organisations, such as large companies. In this work, we propose FieldVMC: a generalisation of VMC, founded on the field-based approach known as aggregate computing, which is applicable to arbitrary topologies (i.e., undirected graphs rather than trees) and supports asynchronous and decentralised execution. We redesign VMC as a field-based computation, hence enabling the emergence of organisational hierarchies out of self-organising interactions among local entities. The benefits of our approach are manifold. Being decentralised and free from topological constraints, our approach makes VMC applicable to arbitrary networks; being based on a well-known computational model, inheriting scalability, asynchronicity, and self-organising capabilities; being implemented in a functional field-based computation framework, fostering reuse and composability. To support our claims, we conduct in-silico quantitative experiments comparing FieldVMC with the original VMC. The results demonstrate that FieldVMC is a monotonic extension of VMC, offering (i) faster convergence, and (ii) enhanced capabilities for capturing, analyzing, and engineering novel phenomena.

Keywords Pattern formation · Growth regulation · Vascular morphogenesis controller · Self-organisation · Plant-inspired computing · Aggregate computing

Introduction

In the field of nature-inspired computing and modular robotics [1], the concept of vascular morphogenesis controller (VMC) has been proposed [2] to model the growth of artificial structures over time. Stemming from a botanical metaphor—the morphogenesis of plants [3]—, the model captures the dynamics of any tree-shaped system whose overall hierarchy is emerging from the self-organisation of elementary units, subject to both local interactions and environmental constraints.

Essentially, VMC entails a distributed and self-organisingfss protocol, where each node in a tree-shaped structure is responsible for (i) the distribution of "resources" (e.g., funding) to its children, and (ii) the collection of "success" (e.g., profits) towards its parent. Combined with local node-creation and -destruction rules, if adequately tuned, the protocol leads the system toward a stable hierarchy. Simulations and experiments involving VMC have been conducted to study if and under which conditions a given organisation is able to grow and sustain itself over time, and how environmental conditions (related to, e.g., the *spatial* distribution of favourable or adverse conditions) may impact such growth.

Scope and intended applications

VMC targets self-organising systems that must evolve or adapt their structure and distribute resources to their components in order to resiliently solve tasks in changing environments. Often, for its operation, a system must locally transport, split, and consume a conserved quantity (e.g., energy, information) while preserving connectivity and

✉ Angela Cortecchia
angela.cortecchia@unibo.it

Giovanni Ciatto
giovanni.ciatto@unibo.it

Roberto Casadei
roby.casadei@unibo.it

Danilo Pianini
danilo.pianini@unibo.it

[1] Department of Computer Science and Engineering, Alma Mater Studiorum-Università di Bologna, Via dell'Università 50, Cesena, 47522 Bologna, Emilia-Romagna, Italy

Published online: 05 December 2025

robustness to asynchrony and *churn* (the situation characterised by frequent joining, leaving, and failing of nodes). Typical application domains include:

- *swarm robotics*—e.g., area coverage and servicing with energy constraints and depots [4, 5];
- *modular/soft robotics*—e.g., where self-reconfiguration of the robot ensemble can support tasks such as locomotion, manipulation, or load distribution [6–8].
- *wireless sensor networks*—e.g., for energy-aware data collection to one or more sinks [9, 10];
- *computer networking*—e.g., for resilient maintenance of peer-to-peer overlay networks over churn [11].

In these settings, the VMC resource to distribute corresponds to battery energy, data rate, or task load; success encodes downstream demand or utility; vessels implement distributed routing/assignment. Indeed, in recent years, VMC has been applied to diverse contexts, including guiding the growth of artificial [12], bio-hybrid [13], and biological plants [14]; self-assembly [15], adaptive path formation [16], and collective locomotion [17] of robots; collective decision-making [18], collective resource distribution [19]; growth and evolution of human organisations [20] like, e.g., structured businesses.

Limitations of VMC

A thorough analysis of VMC (and its related literature) reveals two major limitations, related respectively to the network structure and the algorithm application. First and foremost, VMC assumes a *tree* structure, here intended as a directed acyclic graph with a single root and with a single path connecting the root to each leaf. In organisational terms, this implies that the organisation is strictly hierarchical, therefore neglecting the possibility for two disjoint organisations to merge, or for communication to occur between sibling sub-organisations. Second, VMC is *implicitly synchronous*, as it assumes that (A1) the evaluation of the nodes must proceed from the leaves to the root (and back), and (A2) the update of the whole tree occurs atomically.

Although in some contexts these assumptions are acceptable, in the general case they may induce (possibly hidden) abstraction gaps when VMC is used to model real-world systems, and, at the same time, limit the applicability of the pattern. For instance, in the context of business organisations, assumption (A1) implies that communication of profit at small units is always performed before funding is distributed by the parent unit; and assumption (A2) entails that all units produce their profit and distribute their funding simultaneously. From an engineering perspective, implementing VMC in a real system would require synchronisation mechanisms to ensure that leaves compute only after the root has computed, which can be challenging and imposes a performance burden, as the evolution of the system would be limited by the slowest path from leaf to root (typically, the longest from leaf to root).

Contribution To address these limitations, in this work, we propose FieldVMC: a generalisation of VMC as a *field-based computation* [21–23], in the spirit of the *Aggregate Programming (AP)* paradigm [24]. This generalisation supports arbitrary topologies (i.e., undirected graphs rather than trees), in a truly decentralised and asynchronous way—hence enabling the emergence of organisational hierarchies out of self-organising interactions among local entities. This, in turn, extends the applications of VMC to the study of complex phenomena involving splitting, merging, and disruption of (possibly very large) organisations and facilitates the adoption of the pattern for engineered morphogenetic systems. Last but not least, methodologically, the approach enables addressing VMC-like designs by a *programming language perspective* and in terms of reusable and configurable macro-level blocks of self-organising behaviour [25, 26].

To validate our approach, we exercise FieldVMC with quantitative experiments that demonstrate that it is a monotonic extension of VMC by reproducing some of the results from the literature in a simulated environment, and by showing how our approach can now capture a larger set of phenomena.

Summarising, the contributions of this paper are the following.

- A novel field-based generalisation of VMC model (FieldVMC) that reinterprets VMC within the aggregate computing paradigm, enabling asynchronous and decentralised execution over arbitrary topologies through local, self-organising interactions.
- An implementation of FieldVMC in a functional Kotlin Multiplatform framework, enabling cross-platform execution and promoting code reuse, composability, and type safety.
- A formal proof that FieldVMC is a *monotonic extension* of VMC, as it faithfully reproduces the original model's behaviours while enabling new phenomena such as self-integration, self-division, and self-optimisation.
- A formal proof that FieldVMC is *self-stabilising* under eventual quiescence.
- A set of in-silico experiments showing that FieldVMC is an extension of VMC: it faithfully reproduces the original model's behaviours while enabling new phenomena such as self-integration, self-division, and self-optimisation.
- A scalability evaluation of FieldVMC, realised by measuring the message size and the overall data rate required to sustain the algorithm under different resource availability and configuration.

- An open-source software artefact [27] comprising the full implementation and experimental setup, fostering reproducibility and future extensions.

This manuscript revises and significantly extends the short conference paper [28]. Specifically, we have developed several *new experiments* (cf. type A experiments), extended existing experiments (cf. type B experiments) with *new quantitative evaluation*, and added new material and discussion regarding the system, the programming model, the implementation, the experimental setup, and the results.

Structure of the paper
The remainder of this paper is organised as follows.

In Sect. "Background", we provide a brief overview of VMC and we recall the foundations of the field calculus (FC), as well as basic notions from the AP paradigm. In Sect. "Aggregate vascular morphogenesis controller", we introduce our FieldVMC model, and we describe its implementation based on the *Collektive* framework—i.e., a field-based aggregate programming framework written in Kotlin Multiplatform. In Sect. "Evaluation", we present our experimental setting, and the results we obtained. In Sect. "Limitations and threats to validity", we discuss limitations of the work and threats to validity. In Sect. "Related work", we present a map of related research. Finally, in Sect. "Conclusion", we draw our conclusions and we outline some future work.

Background

Here we recall the main notions of vascular morphogenesis controller, field calculus, and aggregate programming, upon which we build the contribution of this manuscript.

The vascular morphogenesis controller

The seminal work by Zahadat et al. [2] introduced the vascular morphogenesis controller (VMC) model, a plant-inspired approach to modelling the growth of artificial structures over time. The model comes with an algorithm capturing the dynamics of resource distribution in tree-shaped structures, where branches compete for resources in order to grow.

Preliminaries
In VMC, the growable structure is represented as a tree $\mathcal{T} = \langle \mathcal{N}, \mathsf{parent} \rangle$, where each $n \in \mathcal{N}$ is a node and $\mathsf{parent} : \mathcal{N} \to \mathcal{N}$ is the function mapping each node to its parent—except for the root, which has no parent. We denote by $\mathsf{children}(n) \mapsto \{x \mid \mathsf{parent}(x) = n\}$ the set of children of a node n. We say that a node is a *leaf* if it has no children (i.e., $\mathsf{children}(n) = \varnothing$).

Backward-propagation of success
VMC assumes that the nodes of a structure perceive and absorb *success* (S) from the environment, and propagate it towards the root, via their parents. There, 'success' is a generic term denoting some positive income the nodes receive from the environment, propagate to their parents, and, altogether, want to maximise. Reasonable examples of success in concrete applications of VMC can be the amount of light (each leaf of) a plant receives, or the amount of profit a business (unit) produces through its activities.

To model success acquisition, VMC assumes that each node n is associated with a set of sensors $\mathsf{sensors}(n)$, which let the node collect success from the environment. We denote by $\mathbf{I}_s^t \in \mathbb{R}_{\geqslant 0}$ the input gathered by a sensor s at time t. In turn, each sensor $s \in \mathsf{sensors}(n)$ is associated with a weight ω_s, regulating to what extent the perceived success contributes (positively or negatively) to local *success* of node n. Accordingly, at any given time instant $t \in \mathbb{N}$, each node n acts as a collector of success, coming from either the environment or from its children (if any). There, the overall success $S_n^t \in \mathbb{R}_{\geqslant 0}$ collected by n at time t is computed as:

$$S_n^t = \begin{cases} \omega_n + \sum\limits_{s \in \mathsf{sensors}(n)} \omega_s \cdot \mathbf{I}_s^t, & \text{if } \mathsf{children}(n) = \varnothing \\ \mathsf{g}\left(\rho_n + \sum\limits_{s \in \mathsf{sensors}(n)} \omega_s \cdot \mathbf{I}_s^t\right) & \\ \sum\limits_{c \in \mathsf{children}(n)} S_c^t, & \text{otherwise} \end{cases} \tag{1}$$

where $\omega_n \in \mathbb{R}_{\geqslant 0}$ represents the constant production rate of a leaf, $\rho_n \in \mathbb{R}$ is a constant transfer rate, and $\mathsf{g} : \mathbb{R} \to [0, 1]$ is a sigmoid function of choice, mapping the input to the range $[0, 1]$.

Notice that the computation of S_n^t is *recursive*, and that computing the success collected by the root of the tree implies a depth-first, post-order traversal of the tree itself.

Forward-propagation of resources
VMC also captures the propagation of *resources* from the root to the leaves of the structure, stepping through all the internal nodes. There, 'resource' is a generic term denoting any income which is necessary to keep (a portion of) the structure alive. In the plant metaphor, this can be used to model the amount of nutrients that a plant receives from the soil, whereas in the business metaphor, it can be used to model the amount of funding that a composite business unit distributes to its component units.

The core idea behind VMC is to build a positive feedback loop: nodes providing more success *to* the root should be provided with more resources *from* the root, so that they can grow more—and hopefully produce even more success in the future. To capture this aspect, VMC introduces the notion of *vascular pathways* (among the nodes), which regulate the

allocation of resources, and are influenced by the collection of success.

To model vascular pathways, VMC assumes each node n is assigned with a *vessel* connecting it to its parent. The thickness of that vessel $V_n^t \in \mathbb{R}_{>0}$ regulates incoming resource distribution for note n. In other words, the thicker the pathway V_n^t is at time t, the more resources a node n will receive from its parent (and hence spread to its children, in turn) at that time.

Formally speaking, resource distribution works as follows for any non-root node n at time t:

$$R_n^t = R_{\mathsf{parent}(n)}^t \cdot \frac{V_n^t}{\sum_{c \in \mathsf{children}(n)} V_c^t}, \quad \forall n \neq r \tag{2}$$

In other words, each node n receives $R_n^t \in \mathbb{R}_{>0}$ resources at time t, a portion of the resources $R_{\mathsf{parent}(n)}^t$ available at its parent, proportional the ratio between the thickness of its vessel V_n^t, and the sum of the thicknesses of all its siblings' vessels V_c^t.

Of course, the overall effect of resource distribution depends on the resource $R_r^t \in \mathbb{R}_{>0}$ available for the *root* node r at time t. Most commonly, R_r^t is assumed to be constant over time, but it may also be subject to variations. In case that $R_r^0 \equiv R_r^t$, $\forall t$ (i.e., the root perceives a constant amount of resource over time), the amount R_r^0 is a parameter of the model.

Finally, similar to the success S_n^t, we notice that the computation of R_n^t is *recursive*, and that it implies a depth-first, pre-order traversal of the tree.

Growth and reduction

To allow for the structure to grow towards the most favourable conditions, VMC prescribes that the thickness of the vessels connecting the root to the children bringing more cumulative success should be increased over time; and therefore updated according to rule:

$$V_n^{t+1} = \begin{cases} \min(\, S_n^t \,,\ (1-c)\cdot V_n^t + \beta \\ \quad +\alpha \cdot (S_n^t - V_n^t)\,) & \text{if } S_n^t \geq V_n^t \\ \max(\, S_n^t \,,\ (1-c)\cdot V_n^t\,) & \text{otherwise} \end{cases} \tag{3}$$

where $c \in [0, 1]$ is a constant consumption rate of the vessel, $\beta \in \mathbb{R}_{>0}$ is a constant competition rate, and $\alpha \in [0, 1]$ is the adaptation rate of the vessel thickness.

Most notably, the initial thickness $V_n^0 \in \mathbb{R}_{>0}$ of each vessel is a parameter of the model. Notice that the update rule above implies a *synchronisation point*: all nodes must update their vessels *simultaneously*. This is a key limitation of VMC, which we overcome in our FieldVMC model, as described in Sect. "Aggregate vascular morphogenesis controller".

Nodes creation and deletion

To model the growth and shrink of the artificial structure, in VMC each *leaf* node n may either spawn new children or die at time $t+1$, depending on the amount of resources R_n^t it receives at time t. Many strategies are available for implementing the creation (or deletion) of new nodes. The simplest and most common one is to adopt a threshold-based strategy: if R_n^t is above a certain threshold $\theta_{spawn} \in \mathbb{R}_{>0}$, the leaf will spawn a new child at time $t+1$, otherwise, if it is below a certain threshold $\theta_{die} \in\ ]0, \theta_{spawn}[$, the leaf will be removed from the tree. Variants may include probabilistic strategies, or strategies involving the creation of several children at once. The key point here is that, at the model level, the overall structure $\mathcal{T}$ is time-dependent too—hence we write $\mathcal{T}^t$. Implementers would need to define the rule(s) for the creation and deletion of nodes. In the general case, we denote such updates as $\mathcal{T}^{t+1} = \mathsf{f}(\mathcal{T}^t)$, where f is a function mapping the current structure to the next one. In practice, f may consist of partial function(s) defined at the node level.

Implementation

Summarising, a VMC implementation requires: (i) the success perception $\mathbf{I}_s^t$ and the weights ω_s of each sensor s, of each node n, at any time t; (ii) the production (resp. transfer) rate ω (resp. ρ) of the internal nodes (resp. leaves); (iii) the sigmoid function g for internal nodes; (iv) the consumption, competition, and adaptation rates c, β, and α of the vessels; (v) the initial thickness V_n^0 of all vessels (one per node n); (vi) the overall resource R_r^t available to the root node r at time t; and, finally, (vi) the structure update strategy f.

To simplify implementations and to foster reusability, in Table 1, we summarise the parameters of VMC, as well as their domains, dimensionalities, and default values.

Aggregate programming and field calculus

Aggregate programming [24] is a functional macroprogramming approach [25], based on the abstraction of *computational field* [21, 22, 29], in which self-organising behaviour of networks of devices is expressed as composition of functions operating over fields. To understand how it works and its benefits, it is essential to consider three main aspects: its system model, execution model, and programming model.

System model

In AP, the system consists of a network of inter-communicating *devices*, possibly dislocated into a metric space. Spatiality is captured by the notion of closeness: each device has a *neighbourhood* of devices it can directly communicate with, typically via message-passing. Devices are equipped with *sensors* and *actuators* that respectively perceive the local

Table 1 Parameters of VMC, and their description, number of instances, domain, and default value

Param.	Description	How many	Domain	Default
$\mathbf{I}_s^t$	Success perception function of sensor s	$\forall s, \forall n$	$\mathbb{N} \to \mathbb{R}_{\geqslant 0}$	Arbitrary
ω_s	Weight of sensor s	$\forall s, \forall n$	$\mathbb{R}_{\geqslant 0}$	1
ω_n	Production rate of node n	$\forall n : \mathsf{children}(n) = \varnothing$	$\mathbb{R}_{\geqslant 0}$	1
ρ_n	Transfer rate of node n	$\forall n : \mathsf{children}(n) \neq \varnothing$	$\mathbb{R}_{\geqslant 0}$	1
g	Sigmoid transfer function of internal nodes	1	$\mathbb{R} \to [0, 1]$	$\max(0, \min(1, \cdot))$
α	Adaptation rate for the vessels	1	$[0, 1]$	0.9
β	Competition rate for the vessels	1	$[0, 1]$	1
c	Consumption rate for the vessels	1	$[0, 1]$	0.1
V_n^0	Initial thickness of the of node n	$\forall n \neq r$	$\mathbb{R}_{\geqslant 0}$	Arbitrary
R_r^t	Resource perception function of the root	1	$\mathbb{N} \to \mathbb{R}_{>0}$	Arbitrary
f	Structure update function	1	$\mathcal{T} \to \mathcal{T}$	Arbitrary

environment (i.e., information from the device itself or from the device's surroundings) and act upon it.

The main idea of AP is to let the overall system reach a global, *collective* configuration and possibly sustain such configuration over time in response to perturbations. In practice, the target configuration goal depends on the specific application. It may, for instance, consist of a situation where devices converge on the most adequate trajectory among two interesting points, or a situation where devices reach a stable hierarchical structure.

Execution model

One key peculiarity of AP is that it makes the system reach its target configuration through the self-organising behaviour of the devices. To do so, it prescribes that devices are equipped with a *local controller*, i.e., a program governing their behaviour. The program, which is also called an "aggregate program", is the same for all devices[1] [24]. All devices keep executing the same program in rounds, indefinitely. Although no synchronisation is assumed and drifts are well tolerated, devices are normally assumed to execute the program at comparable rates. At each iteration, each device performs three steps:

1. *sense*: the device collects messages coming from its neighbours, sensor readings, and (if available) the state produced by itself in the previous iteration;
2. *compute*: the device executes the aggregate program, producing an output value, the messages to be sent to each neighbour, and the state to be stored for the next iteration; and, finally
3. *interact*: the device sends messages to its neighbours, and actuation is performed.

Typically, only the last message from a neighbour is considered, and messages have a duration of validity (i.e., they may expire). Depending on the actuators installed on a specific device, the aggregate program may trigger actions including moving the device itself, disconnecting it to leave the system, or (when meaningful) even creating new devices. In all such situations, the neighbourhood of the acting device may change, hence affecting the communication topology and, ultimately, the overall behaviour of the system. This last assumption, although acceptable and adopted in most cases, can be relaxed if a reactive approach is adopted instead [30, 31].

Programming model

In general terms, the essence of the aggregate programming model is captured by a class of functional languages collectively known as FC [21]. These provide constructs for conceptually manipulating distributed data structures called *computational fields* (or just "fields" for short), i.e., maps from devices (or also, the corresponding locations, for a spatial interpretation) to values. Consider that fields of values can work as a denotation for "collective" inputs or outputs: for instance, the temperature of a room can be denoted as a field of numbers corresponding to the individual temperatures sensed by the devices in that room; similarly, a field of vectors can denote a collective command to a swarm of robots that are programmed to explore some environment.

Although a complete description of the principles and primitives of the FC is beyond the scope of this paper, in this section we provide a brief overview of a selection of

[1] To be precise, in the higher-order field calculus (HOFC) [29] is allowed for aggregate programs to build fields of aggregate functions. These, in turn, can be exchanged among devices and executed, creating de-facto a situation where devices may run different programs. However, the core aggregate program orchestrating the execution of these sub-programs must remain the same for all devices.

concepts that will be used in the contribution presented in Sect. "Aggregate vascular morphogenesis controller". We present them using the same Kotlin syntax that we will use in Sect. "Aggregate vascular morphogenesis controller" to implement FieldVMC.

Fields (local vs. global interpretation)
Any local value can be interpreted as a field. For instance, program fragment

```
7.5
```

can be interpreted as usual if we focus on a single device. However, since that program fragment is run (in a sense–compute–interact loop) by every device that is part of the aggregate system, it can also be interpreted as a constant (not changing over time) and uniform (same value in different devices/spatial locations) field of floats, holding `7.5` everywhere. If we generate a random value,

```
kotlin.random.Random.nextInt(0, 100) // Produces a different value in each round
```

then we will have, generally speaking, a non-constant (since the default random generator in Kotlin is initialised with a random seed) and non-uniform (since scheduling is generally not synchronised) field.

Therefore, we can conceptually denote fields using local values. It makes sense, since a "global" field is not easy to operationalize in a distributed system. At the language level, the only type of field that is reified is the "neighbouring field"—globally, it is meant as a field of (neighbouring) fields.

Neighbouring fields: observation of neighbouring values
For any variable `x` local to a device, the value of the same variable in neighbouring devices can be observed using **`neighboring`**`(x)` [2]. **`neighboring`** is a function generic in some type `T`, accepting a value of type `T` as input, and returning a field mapping each neighbouring device identifier (ID) to a value of the same type.

```
fun <T> neighboring(local: T): Field<T>
```

For example, in a network of three devices forming a line network where a device `id1` is connected to `id2`, and `id2` is connected to `id3`, the following program:

```
println(neighboring(localId))
```

would print:

- in `id1`: ϕ`(local: id1=>id1, neighbors: [id2=>id2]);`
- in `id2`: ϕ`(local: id2=>id2, neighbors: [id1=>id1, id3=>id3]);`
- in `id3`: ϕ`(local: id3=>id3, neighbors: [id2=>id2]);`

where the ϕ symbol is used to denote that the value is of a field type, and where, for convenience of representation, the local values is kept separate from neighbours' values.

Evolution in time and space The majority of meaningful aggregate computations are typically "progressive", meaning that, given constant inputs, multiple rounds of computations and communication are needed to produce correct and stable outputs. Additionally, when inputs change, the changes require multiple rounds to be absorbed towards new correct outputs. This idea of collective values that change over time is also called *field evolution*. Evolving a field locally can be achieved by applying a function to a value produced the previous round. Recent versions of FC combine this with communication [32] into a single construct (since, often, the locally evolved value is also communicated to neighbours).

Therefore, to evolve a field through time and space, AP uses **`exchanging`** [32], which models stateful communication among neighbouring devices. **`exchanging`** is a function generic in `S` (the type of data being shared) and `R` (the type of data being returned). It takes a local value of type `S` and a function operating over a `Field<S>`, producing a `Pair<Field<S>,R>` of a field of the same type and a return value of type `R`. The produced field is used to decide which information to store and send: the field's local value is stored as the next round's local value, while every neighbouring device receives the value in the field corresponding to its identifier. If unknown devices become neighbour, they are sent the default `initial` local value.

[2] We denote FC primitives in red, Kotlin keywords in blue, and derived (library) constructs in purple.

```
fun <S, R> exchanging(initial: S, body: (Field<S>) -> Pair<Field<S>, R>): R
```

To beautify and clarify the code, a function **yielding** is provided, which is a syntactic sugar for building a pair of a field and a return value.

```
fun <S, R> Field<S>.yielding(body: () -> R): Pair<Field<S>, R>
```

We present a simple usage example of this primitive through the construction of a self-healing *gradient* field [26] mapping each device to its hop-count distance estimation from the nearest device satisfying a certain condition. In the following snippet, we implement a self-healing variant of the Bellman–Ford algorithm, commonly used in distributed systems to estimate the hop-distance [26].

```
import Double.POSITIVE_INFINITY as infinity
fun bellmanFord(cond: Boolean): Double = exchanging(infinity) { distances: Field<Double> ->
    val hopDistances: Field<Double> = distances + 1.0
    val minDistance = hopDistances.minValue(base = infinity)
    hopDistance.yielding { if (cond) 0.0 else minDistance } // Pair<Field<Double>, Double>
}
```

Notice that, though the return value is purely local (`Double`), we still colloquially refer this function as producing a field—denotationally given by the set of local values, one for each device running the computation. Indeed, aggregate computations are amenable to a dual interpretation: local (what happens in a single device) or global (what collectively happens to an ensemble of devices).

Domain segmentation

The reader can notice that in the previous snippet we computed `minDistance` seemingly uselessly outside of the **if** expression. In reality, computing it in the **else** branch would compromise correctness. In fact, branching in AP implies *domain segmentation*: devices taking different branches belong to different computation domains and hence do not communicate (i.e., do not see each other as neighbours). In our Kotlin implementation, we used a compiler plugin to extend the semantics of **if**/**else** (hence shown in red) when fields are involved, effectively supporting domain segmentation. In the example, thus, the devices where `cond` holds (namely, the sources of the gradient) would not communicate with the others, preventing the "bootstrap" of the gradient computation: a field of `infinity` values would result everywhere but in devices where `cond` holds, which would return `0`.

Toolkits

The AP paradigm is general-purpose: it can be applied to a wide range of applications, from small to large-scale scenarios, from in-silico experiments to internet of things (IoT) applications. Several implementations of AP exist, including Protelis [33] (a standalone domain-specific language (DSL) [34] developed in Xtext [35] and targetting the Java platform), ScaFi [36] (a Scala-based internal DSL [37]), and FCPP [38] (a DSL embedded in C++), each implementing a slightly different variant of the original calculus, and with mixed pros and cons.

In this work, we leverage an experimental Kotlin-multiplatform [39] implementation of the eXchange calculus [32] called *context receivers*.[3] The reasons for this specific choice are threefold: (i) first-class Java virtual machine (JVM) support (we thus excluded FCPP, based on C++); (ii) static type system (which ruled out Protelis); and (iii) native support for the `exchange` primitive (currently unavailable in ScaFi), called **exchanging** in our implementation. Technically, the expressiveness of the latter is useful—beyond previous proposal for neighbouring communication primitives—because it supports the sending of different values to different neighbours.

Summary

A VMC reformulation within the framework of AP natively overcomes the limitations of the original VMC discussed in Sect. "The vascular morphogenesis controller", as it computes over undirected graphs (arbitrary network topologies), and has no implicit synchronisation mechanism. Additionally, it is an established approach for the construction of *composable* self-organising systems, as it enables reuse via familiar programming mechanisms such as functional com-

[3] https://github.com/Collektive/collektive.

position. Thus, an AP-native version of VMC would simplify the reuse of the algorithm in different contexts, paving the way to the usage of the pattern in practical applications.

In the next section, we discuss how to realise a VMC within the AP framework.

Aggregate vascular morphogenesis controller

To characterise our proposal, FieldVMC, we need to map VMC's concepts to the elements of the aggregate programming model (Sect. "Model mapping"). Then, we discuss the design and implementation of the self-organising processes supporting the morphogenetic capabilities of the approach (Sect. "Implementation").

Model mapping

In FieldVMC, a node represents an *agent* (of an organisation or a morphogenetic structure); a neighbouring link denotes a *communication link*, namely the possibility of two agents to communicate; and each agent is assumed to have the following sensors:

- `success`: provides a value (e.g., a real number) denoting the local success provided by the environment (e.g., light intensity for plants, or profits for organisational entities);
- `resource`: provides a value denoting the amount of resource locally available at the agent (e.g., nutrients in plants, or budget in organisations);
- `position`: provides a value denoting the position of the agent in space;
- `distance`: provides a map of values denoting the distance of the agent from its neighbours;

and the following actuators:

- `spawn` : creates new agents;
- `destroy`: self-destructs the agent.

Variants of the pattern can be realised with one or no actuator, in which case the structure will be unable to grow, shrink, or both. Also, `position` is not strictly necessary, especially if the system has no `spawn` actuator, or if the node creation criteria is not based on spatial proximity. `distance` can be implemented in several ways, the simplest being a hop-count distance, which can be implemented in *Collektive* as **`neighboring`**`(1)` (a function returning a `Field` mapping each reachable device to `1`).

Implementation

In FieldVMC, we support morphogenesis in terms of multiple self-organisation processes. Since we start from a graph, we need to construct the tree overlay structure supporting the forward and backward flows of success and resources. Crucially, we extend over VMC by supporting *multiple trees* and a *dynamic, resilient set of trees*. To achieve this, we define the overall design as an instantiation of the *self-organising coordination regions (SCR)* pattern [40], which is a flexible way to decompose complex problems distributed in space. SCR addresses problem decomposition and task assignment in distributed settings by: (i) electing sparse leaders; (ii) expanding regions from leaders; (iii) creating upstream information flows [41] towards the leader; (iv) perform decision-making at the leader and propagate decisions using downstream information flows.

```
1  interface SuccessSensor { fun success(): Double }
2  interface ResourceSensor { fun resource(): Double }
3  interface LocationSensor { fun position(): Pair<Double, Double> }
4  interface DistanceSensor {
5    fun <ID> Aggregate<ID>.distance(): Field<ID, Double>
6  }
7  interface DeviceLifecycle {
8    // May spawn new agents, do nothing, or destroy itself
9    fun Aggregate<ID>.manageLifecycle(potential: Double, resource: Double): Unit
10 }
```

Listing 1 The sensors and actuators used in the implementation of FieldVMC.

```
1  context(SuccessSensor, ResourceSensor, LocationSensor)
2  fun Aggregate.vmc(isRoot: Boolean): Double {
3    // Leader area construction
4    val potential = distanceTo(isRoot)
5    // Upstream information flow
6    val success = convergeCast(potential, success(), Double::plus)
7    // Downstream information flow
8    val localResource = weightedGradientCast(
9      potential = potential,
10     local = if (isLeader) resource() else  0.0,
11     weight = success
12   )
13   // Actuation
14   manageLifecycle(potential, localResource)
15   return localResource
16 }
17
18 context(SuccessSensor, ResourceSensor, LocationSensor, DistanceSensor, DeviceSpawn)
19 fun Aggregate.vmc(radius: Double): Double =
20   // Leader election
21   vmc(boundedElection(resource(), radius) == localId)
```

Listing 2 The `vmc` functions are the entrypoints of FieldVMC, parametrised either on a `Boolean` value (for statically-defined roots) or on a maximum distance to be used for electing roots collectively.

Listing 2 shows the *Collektive* code implementing the VMC logic by adhering to the SCR pattern. In the following, we explain this code, and provide details on the involved functions. For the sake of conciseness, we do not deep-dive into all the technical details of the complete implementation; instead, we release a complete working implementation as open-source with a permissive licence: we recommend the reader interested in reusing the approach to begin from there.

General structure

The implementation of FieldVMC in Listing 2 follows the SCR pattern: (i) leader election (Lines 20–21); (ii) area creation (Lines 3–4); (iii) upstream communication (Lines 5–6); (iv) downstream communication (Lines 7–12); after which a decision is made on whether or not to spawn new agents or abandon the system. To support cases in which the roots are statically defined, we overload the `vmc` function: the first overload expects a `Boolean` value representing whether or not the current node is a root. It can be provided directly, or a leader election process can be used instead; in the latter case, a second overload of `vmc` expects instead the maximum extension of the leader area as a `Double`. In our implementation, we leverage an experimental Kotlin feature called *context receivers*[4] which, roughly, make the function visible only where an instance of a certain type is contextually available as a receiver. In our implementation, we use them to specify which sensors and actuators are required to execute the function (Lines 1 and 18). Their declaration is summarised in Listing 1, while their implementation depends on the specific use case (in the artefact repository, we provide implementations for the simulation platform used for experimentation).

Roots identification

Assuming the most general case, we begin with a dynamic identification of the root(s) of the tree(s) through a *leader election* process. Although any (multi-)leader election algorithm is suitable, in this work we use *Bounded Election* [42, 43], a priority-based multi-leader election algorithm, as it is proven to be self-stabilising. *Self-stabilisation* [44] is the property of a distributed system to converge, in finite time, to a correct and stable configuration regardless of its initial conditions. This entails *resilience*: the failure of a leader will be compensated by the election of another one. The algorithm is also *priority-based*, which, in the case of FieldVMC, we leverage to elect as a root the node which is perceiving the highest amount of resources (cf. `resources()`). Finally, it can elect multiple leaders provided a maximum distance after which a new leader is elected. The value for this parameter depends on the specific application.

Area expansion

Once the roots are identified, we need to select the sub-graph around each leader where the tree structures will get built. This is done by building a `potential` field, denoting, for

[4] Context receivers have been proposed in KEEP-259 (archived at https://archive.is/YmZ8r) and were available in the latest Kotlin compiler available at the time the research has been conducted. Context receivers will never become a stable feature of Kotlin, they will likely be replaced by a very similar feature called *context parameters* (KEEP-367, see https://archive.is/twmFC). Future implementors of the pattern can refer to this new feature for a more up-to-date implementation.

each agent, the distance from the closest root. This kind of structure is also known as gradient [26, 45, 46], and it is a well-known building block common in bio-inspired computing systems [47], (including AP [26]), and frequently used in networking (for instance, see the Bellman-Ford algorithm mentioned in Sect. "Programming model").

Upstream flows

Once a potential field is established, it can be leveraged to propagate information upstream to the root by having the information *descend* the gradient. This algorithm is known as converge-cast and, similar to the gradient, it is a well-known building block in AP-based systems [26] for which several implementations exist [48]. Among them, one of the most commonly used builds a spanning tree dynamically by selecting, for each node, the neighbour with the lowest potential. In our case, this is the behaviour of the `convergeCast` function in Listing 3, which builds the underlying spanning tree (using the `findParent` function of Listing 4 internally); sends the local information to the leader (in this case, the success perceived by the agent through a call to `success()`); and then aggregates it with the provided function (in this case, by summing all values with `Double::plus`).

Downstream flows

Each node uses the success perceived by its children (if any) to forward a certain amount of resources to them. This operation is akin to a gradient cast, with the difference that the information must be diffused anisotropically, i.e., different information must be sent to different neighbours. In our case, we used the implementation shown in Listing 5, where we first find the parent of all neighbours, including self (Lines 4–5). Then we compute the amount of locally available resource (Lines 5–9), using the provided value if we believe we are the root, or the value received from the parent, otherwise. Once we know how much resource we can send, we fetch for success (`weight`) in neighbouring nodes, keeping only the value for those that are our children (Lines 10–11). We compute the total success at Line 12, and then split the resource (`toSpread`) based on the success of each child (Lines 13–15). We send away this value (Line 16), and finally we return the local amount of resource (Line 17). In the proposed implementation, the thickness of the vessels is not explicitly computed. Variations accounting for limits to the amount of resources that can be sent or different weighting strategies can be implemented by modifying the `weightedGradientCast` function.

```
1  fun <T, ID : Comparable<ID>> Aggregate<ID>.convergeCast(
2      potential: Double,
3      local: T,
4      disambiguateParent: (ID, ID) -> ID = { a, b -> minOf(a, b) },
5      reduce: (T, T) -> T,
6  ): T = share(local) { field ->
7      val parent = findParent(potential, disambiguateParent)
8      val neighborParents = neighboring(parent) // Each device is mapped to its parent
9      val childrenValues =
10         neighborParents.alignedMap(field) { itsParent, itsLocal ->
11             Channel(isFromChild = itsParent == localId, itsLocal)
12         }
13     childrenValues.fold(local) { accumulator, channel ->
14         if (channel.isFromChild) reduce(accumulator, channel.localValue) else accumulator
15     }
16 }
```

Listing 3 The function building the spanning tree and sending the local success to the root.

```
1  fun Aggregate.findParent(
2    potential: Double,
3    disambiguateID: (ID, ID) -> ID = { a, b -> minOf(a, b) },
4  ): ID {
5    val neighboringPotential: Field<Double> = neighboring(potential)
6    val localMin = neighboringPotential.min() ?: potential
7    return neighboringPotential.asSequence()
8      .mapNotNull { (id, v) -> id.takeIf { v == localMin } }
9      .reduce(disambiguateID)
10 }
```

Listing 4 The function finding the lowest potential value in a locality, used to build the spanning tree.

```
fun Aggregate.weightedGradientCast(
  potential: Double, local: Double, weight: Double
): Double = exchanging(local) { field: Field<Double> ->
  val parent = findParent(potential) // Id of the parent
  val otherParents = field.excludeSelf() // Maps of each neighboring id to its parent id
  val toSpread: Double = when {
    parent == localId -> local // I am the root
    else -> otherParents[parent] ?: 0.0
  }
  val childrenWeights: Field<Double> = neighboring(parent to weight)
    .map { (itsParent, itsWeight) -> if (itsParent == localId) itsWeight else 0.0 }
  val total: Double = childrenWeights.sum()
  val outgoingResources: Field<Double> = childrenWeights.map {
    if (total <= 0) 0.0 else it * toSpread / total
  }
  outgoingResources.yielding { neighboring(toSpread) }
}.localValue
```

Listing 5 The weighted gradient cast implementation: local information is spread to children, dividing it based on weights (in our case, the perceived success)

Analysis of FieldVMC w.r.t. VMC

Here we analyse FieldVMC with respect to VMC. We start by summarising the capabilities of FieldVMC, with a focus on the ones which are not supported by the classic VMC—which can therefore be considered as extensions. We then prove why and under which conditions FieldVMC is—behaviourally speaking—equivalent to VMC, other than self-stabilising. Combined with the aforementioned extensions, we conclude that FieldVMC is a monotonic extension of VMC.

Distinctive features of FieldVMC

Table 2, summarizes the main capabilities of FieldVMC, referencing the particular section of this paper where each capability is introduced.

General topologies Like its predecessor, FieldVMC can operate on tree-like topologies, hence ensuring backward compatibility—meaning that existing FieldVMC behaviours remain valid under the new semantics. However, it also supports graph topologies, such as mesh networks, hence overcoming a core limitation of the original model.

Asynchronous execution FieldVMC tolerates asynchronicity in round execution, while VMC implicitly requires synchronisation. Tolerance to asynchronous execution is is beneficial for both performance and fault tolerance. Although worst-case convergence time is identical in both cases, the average case is generally better under asynchronicity. In fact, sub-portions of the network with shortest paths to the root may stabilise faster, as downstream flows are established and working while other parts of the network are still converging. Besides performance, this robustness also increases resilience and readiness to changes. However, asynchronicity may raise concerns about FieldVMC's guarantees on convergence and stability. Accordingly, we discuss this matter in Sects. "FieldVMC is a monotonic extension of VMC" and "Self-stabilisation under eventual quiescence".

Multiple leaders
The leader election algorithm identified in Sect. "Roots identification" supports the election of multiple leaders, in turn allowing FieldVMC to split large networks into multiple sub-trees, e.g., for easier management. Consequently, FieldVMC's support for multiple leaders generalises the single-root constraint of VMC, enabling the formation of dynamic multi-root structures.

Merge and split
FieldVMC inherits the self-organising traits of AP, so it can operate through temporary segmentation and when previously disjoint networks merge. These dynamic reconfiguration behaviours are not supported by the original VMC, which assumes a single, statically chosen root.

The overall dynamics of a FieldVMC system is configured by the specific implementation of the `DeviceLifecycle` component and the specifics of the root selection / leader election process. However, in general, FieldVMC enables a richer set of dynamics. In particular, assuming a dynamic leader election process based on the amount of resources perceived by nodes, the structure may stabilise into one *or multiple* subtrees depending on the distribution of resources in the environment, whereas in VMC only a single tree can "survive", as the root device is selected statically. Conversely, as multiple sub-trees can co-exist in the same network, they could merge into a single tree—provided that there is a benefit for the resulting structure. Additionally, FieldVMC can

Table 2 Capabilities realised by FieldVMC, the underlying mechanism, where the evidence appears, and whether classic VMC supports them

Capability	Mechanism in FieldVMC	Evidence	VMC
Asynchronous execution on arbitrary graphs	Inherited from the FC semantics	Section "Programming model"	×
Operation on non-tree topologies (cycles, multi-path)	SCR produces a dynamic forest over an arbitrary communication graph	Section "Analysis of FieldVMC w.r.t. VMC"; Listing 3, Listing 5	×
Multiple leaders (multi-root)	Leader identification and bounded election with local tie-breaking	Sections "Roots identification, Distinctive features of FieldVMC and "Experiments of type **(B)**: analysing FieldVMC"	×
Merge of disjoint organisations	Dynamic parent (re)selection; recomputation of upstream/downstream flows under topology changes	Sections Distinctive features of FieldVMC, Experiment B.1: self-integration and 7	×
Self-division/controlled splitting	Local lifecycle thresholds over vessel flux; competitive reinforcement of alternative downstream paths	Sections Distinctive features of FieldVMC, Experiment B.2: self-division and 9	×
Anisotropic downstream allocation on general neighbourhoods	`weightedGradientCast` on undirected graphs; weights from local success/metric fields	Listing 5; Eq. (2)	✓ (on trees)
Modularised growth and shrink rules	`DeviceLifecycle` encapsulates the spawn and destruction policy	Listing 5; Eq. (2)	✓ (configurable, not modular)

handle the failure of the root node, which would cause the entire structure to collapse in VMC. A consequence of the root selection by election is that children spawned into a more resourceful area may cause the parent to lose its leadership, hence optimising the structure to maximise resource collection. This behaviour is analogous to plants: a branch that reaches richer soil may develop roots and become the new main trunk, or split off as an independent plant.

Growth and shrink

In FieldVMC, the growth and shrink of the structure is modelled by the `DeviceLifecycle` component. Different implementations can produce very different final structures, and the strategy to adopt strongly depends on the system under study or development. Reasonable strategies include, for instance, spawning (resp. self-destroying) a node when the resources remain above (resp. below) a certain threshold for a sufficient amount of time, or spawning new nodes based on the count of current children or neighbours, etc. Although most strategies can be implemented in VMC as well, in FieldVMC the growth and shrink strategies are explicit and modular, hence they can be easily customised and replaced.

FieldVMC is a monotonic extension of VMC

Despite all such additional capabilities, one may wonder whether FieldVMC is still able to do what VMC can. Here we prove that this is indeed the case, when (i) FieldVMC is applied to static tree structure (no spawn/destroy is allowed), (ii) it is forced to work synchronously, and (iii) the amount of resource perceived by the root is fixed.

In fact, under the above constraints, the `convergeCast` function in Listing 3 computes the VMC success recursion (Eq. 1), function `weightedGradientCast` (Listing 5) realises the VMC resource split (Eq. 2), and vessel updates require twice the network diameter rounds to stabilise (in the worst case), as every node will need to receive information from all its descendants and then send information to all its children. Therefore, every VMC instance has a behaviourally equivalent FieldVMC program. In the following, we prove this claim formally, using the notational conventions of Sect. "The vascular morphogenesis controller".

Proposition 1 (FieldVMC is behaviourally equivalent to VMC under tree/synchronous assumptions) *Let* $\mathcal{T} = \langle \mathcal{N}, \mathsf{parent} \rangle$ *be a connected communication tree whose* parent *relation accounts for a single root* n_0. *Assume synchronous rounds, fixed root resource input* R_{n_0}, *and no spawn/destroy. Then, the* `convergeCast` *in Listing 3 computes the VMC success recursion of Eq.* (1) *for every node, and* `weightedGradientCast` *in Listing 5 realises the VMC resource split of Eq.* (2).

Hence, after at most $2 \cdot \mathsf{diam}(\mathcal{T})$ *rounds, the success and resource values at all nodes coincide with those given by VMC, and the ensuing vessel update decisions coincide whenever the same thresholds/tie-breaking are used.*

Proof Let $\mathsf{depth}(n)$ denote the depth of some node $n \in \mathcal{N}$ in the tree $\mathcal{T}$—i.e., its distance from the root n_0—, and let us call $H = \max_{n \in \mathcal{N}} \{\mathsf{depth}(n)\}$ the height of the tree.

Let S_n denote the VMC success of node n defined by Eq. 1 (a pure bottom-up recursion on $\mathcal{T}$), and let $s_n[t]$ be the value

locally stored at n by `convergeCast` after t synchronous *up* rounds.

Similarly, let R_n denote the VMC resource available to node n, as given by Equation (2) (a top-down recursion parameterised by $\{S_n\}$) and let $r_n[t]$ be the value locally stored at n by `weightedGradientCast` after t synchronous *down* rounds.

(i) *Upward correspondence.* We prove by strong induction on k that for every node n whose deepest descendant is at distance k, we have $s_n[k] = S_n$.

For $k = 0$ (leaves), both VMC and `convergeCast` use only local terms, hence $s_n[0] = S_n$ by definition of Eq. 1.

Assume the claim holds for all subtrees of height $< k$.

Consider a node n whose deepest descendant is at distance k.

At round k, each child $c \in \mathsf{children}(n)$ has $s_c[k-1] = S_c$ by the induction hypothesis; `convergeCast` at n applies exactly the combining function of Eq. (1) to the multiset $\{s_c[k-1] \mid c \in \mathsf{children}(n)\}$ and its local term, hence $s_n[k] = S_n$.

Therefore, after H up rounds, all nodes hold their VMC success: $s_n[H] = S_n$.

(ii) *Downward correspondence.* Fix the success field at the VMC values just established.

We prove by induction on $d = \mathsf{depth}(n)$ that after d down rounds starting from n_0 with $r_{n_0}[0] = R_{n_0}$, we have $r_n[d] = R_n$.

For $d = 0$ (the root), the claim is immediate.

Assume the claim for all nodes at depth $< d$; take n at depth d with parent $p = \mathsf{parent}(n)$.

At down round d, node p has $r_p[d-1] = R_p$ by the induction hypothesis; `weightedGradientCast` applies Eq. 2 using R_p and the fixed success values to allocate resources to each child, so $r_n[d] = R_n$.

Hence, after H down rounds, all nodes hold their VMC resource: $r_n[H] = R_n$.

(iii) *Stabilisation bound and vessel updates.* Combining (i) and (ii), after H up rounds followed by H down rounds (i.e., at most $2H \le 2 \cdot \mathsf{diam}(\mathcal{T})$ rounds), both success and resource match VMC everywhere.

Since vessel updates are functions of these values plus fixed thresholds/tie-breaking, the resulting edge selections coincide. □

Corollary 1 (Monotonic extension) *For every VMC instance on a rooted tree $\mathcal{T}$ with synchronous rounds and fixed R_r, there exists a FieldVMC program (using `convergeCast` and `weightedGradientCast` with identical local functions and thresholds) whose synchronous execution produces the same success/resource/vessel traces as VMC after at most $2 \cdot \mathsf{diam}(\mathcal{T})$ rounds.*

Moreover, FieldVMC expresses behaviours not definable in VMC, including multi-root operation and execution on non-tree topologies.

Proof The first part is immediate from Eq. (1) by taking the same combining/splitting functions and thresholds.

For strictness, consider any connected graph that temporarily contains two leaders each receiving an exogenous resource input (multi-root), or any graph with a cycle maintained during execution.

VMC requires a single rooted tree at all times, hence cannot realise such behaviours without violating its structural constraints.

Instead, FieldVMC admits them via leader election and dynamic parent selection over arbitrary graphs.

Therefore, the expressible behaviours of VMC are a proper subset of those of FieldVMC. □

Self-stabilisation under eventual quiescence

Under unbounded churn (nodes/links constantly changing), the algorithm cannot self-stabilise in the classical sense [44], because the target "legitimate" state keeps moving. In this section, we thus prove self-stabilisation under eventual quiescence: after any finite burst of faults/topology change, if the graph and inputs stop changing and the devices execute fairly, the system converges in finite time to a legitimate state and stays there.

Proposition 2 *(FieldVMC is self-stabilising under eventual quiescence) Let $(\mathcal{G}_t)_{t \ge 0}$ be a time-indexed sequence of connected communication graphs and let R_t denote the exogenous resource input(s) at the leader node(s). Assume there exists t^* such that for all $t \ge t^*$:*

- (A1) *$G_t = G_{t^*}$ (stable graph) and the set of leaders is fixed, and the node spawning and node destruction capabilities of the algorithm are inactive;*
- (A2) *$R_t = R_{t^*}$ (stationary inputs);*
- (A3) *the round execute fairly (every node executes infinitely often) and messages have finite delay;*
- (A4) *each non-leader node n chooses a parent $\mathsf{parent}(n)$ among its current neighbours so that a* strict *potential decreases along the chosen edge (e.g., gradient distance to the closest leader), with a deterministic tie-break to avoid cycles (e.g., comparison of the nodes' unique identifiers);*
- (A5) *the local combining/splitting functions used by `convergeCast` and `weightedGradientCast` are the same as in VMC (Eqs. 1 and 2) and are applied to the instantaneous neighbourhood state.*

Then, from any system's state at time t^* (arbitrary buffers, parents, and field values), FieldVMC converges in a finite number of rounds to a legitimate state *in which:*

(L1) *the parent pointers form a directed forest rooted at the leaders (no cycles);*
(L2) *the success values at all nodes equal the unique fixed point of Eq.* (1) *over that forest;*
(L3) *the resource values equal the unique fixed point of Eq.* (2) *over that forest; and*
(L4) *vessel selections are stable given those fixed-point values and the deterministic thresholds/tie-breaks.*

Moreover, if D is the hop-diameter of $\mathcal{G}$ and H the maximum root-to-leaf depth of the stabilised forest, then stabilisation occurs in at most $D+2H$ synchronous rounds; in particular $D+2H \leq 3D$.

Proof We argue by phases with a decreasing potential.

Phase 1 (parent/forest stabilisation). By (A4), each non-leader repeatedly selects a neighbour of strictly lower potential (e.g., smaller gradient distance to a leader) with a deterministic tie-break. Since the potential is bounded below at the leaders, no infinite descent is possible; under fairness (A3), each node updates finitely many times until no local improvement exists.

Strict decrease along chosen edges forbids directed cycles, hence the parent pointers stabilise into a directed forest rooted at the leaders.

On a stable topology, standard gradient propagation yields correct leader distances within at most D rounds, so parent choices stabilise within D rounds.

Phase 2 (success fixed point on the forest). With the forest fixed, `convergeCast` implements the bottom-up recursion of Eq. (1) by (A5). Let H be the maximum distance to a root in the forest. A strong induction on subtree height shows that after H upward evaluations all nodes hold the VMC success (base case at leaves; inductive step combines already-correct children). Fairness ensures progress; bounded delay prevents starvation.

Phase 3 (resource fixed point on the forest). Given the now-correct success values, `weightedGradientCast` realises the top-down recursion of Eq. (2) by (A5) from the stationary (A2) inputs. By induction on depth, after H downward evaluations all nodes hold the corresponding resource values.

Phase 4 (vessel stability). Vessel decisions are pure functions of local success/resource and fixed thresholds/tie-breaks. Once those values are fixed, no subsequent updates occur.

Combining the phases, from any initial state at t^* the system reaches a legitimate state satisfying (L1)–(L4) in at most $D+H+H = D+2H \leq 3D$ synchronous rounds (or the analogous number under (A3)).

Thereafter, with (A1)–(A2) maintained, the state remains legitimate. □

Corollary 2 *(Multiple leaders) Under the conditions of Eq.* (2) *with multiple leaders, the argument applies per tree of the stabilised forest; convergence and stability hold simultaneously across all trees.*

Corollary 3 *(Dynamic leader selection) If leaders are selected dynamically, Assumption* (A1) *is violated. However, if the leader selection algorithm is self-stabilising, it will eventually converge to a stable set of leaders, restoring* (A1)*; thus, when combined with FieldVMC, the overall algorithm will be self-stabilising under assumption* (A2)*.*

In this case, if the election algorithm converges in E rounds, the overall system reaches a legitimate state in at most $E+D+2H$ rounds.

Remark (on churn). If assumptions (A1)–(A2) are violated indefinitely (unbounded churn or drifting inputs), the target fixed point moves; the system tracks changes and re-stabilises after each quiescent interval but cannot be globally stable.

Remark (on leader election). Dynamic and self-stabilising leader election algorithms exist. In this work, we relied on bounded election [42], which is a multi-leader election for which self-stabilisation has been proven. Since bounded election stabilises in at most D rounds, our implementation of FieldVMC is guaranteed to stabilise in at most $2D+2H \leq 4D$ rounds.

Evaluation

In this section, we evaluate the proposed FieldVMC approach *empirically*, to demonstrate that it is a *monotonic* extension of VMC. In other words, we show that FieldVMC can be used to model the same phenomena as VMC (namely: self-construction, and -regeneration), plus some more (namely: self-integration, -repair, and -optimisation).

Our demonstration is based on a set of experiments, where both VMC and FieldVMC are simulated in the same *in-silico* environment. Broadly speaking, we perform experiments of three sorts: experiments of the first sort **(A)** aim at proving that FieldVMC supports self-construction and self-healing as VMC does (but more efficiently), whereas experiments of the second sort **(B)** aim at proving that FieldVMC supports self-integration, self-division, and self-optimisation, which VMC does not support by construction, finally, experiments of type **(B)** aim to showcase how the resource requirements of FieldVMC scale in terms of communication costs.

The main difference is that, in experiments of type **(A)**, both VMC and FieldVMC are simulated in similar

experimental setups, and their dynamics are quantitatively compared; whereas in experiments of type **(B)** and **(C)** , only FieldVMC is simulated, and its dynamics is quantitatively analysed in various experimental setups.

Generally speaking, each experiment consists of a parametric simulation, repeated 500 times with a different random seed for each parameters' values combination. Results are then averaged over the different runs. In practice, simulation are implemented using the Alchemist simulator [49]. For the sake of reproducibility, we made the source code of our experiments open-source, by releasing it on GitHub[5] with a permissive license and by archiving it on Zenodo [27] for future reference.

Accordingly, in the remainder of this section, we first present the aspects common to all experiments (Sect. "Experiments setup"), and then we detail each experiment, specifically focussing on experiments of the first (Sect. "Experiments of type **(A)**: FieldVMC vs. VMC") and second sort (Sect. "Experiments of type **(B)**: analysing FieldVMC").

Experiments setup

Our experiments are conducted in a simulated environment, where the nodes are placed in a bi-dimensional Euclidean space. We assume distance-based communication: nodes sufficiently close can exchange information. We place resources in the environment, following a bivariate Gaussian distribution centred in $(10, 0)$, scaled such that the maximum value at the centre (namely, the maximum amount of resources that can be sensed at any time) $\max(R) = 1000$, with a standard deviation $\sigma = 5$. Specularly, at $(10, 20)$, we placed a bivariate Gaussian distributed source of success with the same parameters as the resource source.

We tune the amount of resource and success in such a way that there is enough resource to sustain the growth of a structure reaching the success source. Notice that this is not a requirement for the algorithm to work, instead, it captures a challenging situation: an excess of resources would make it easier to capture a lot of success even without a well-structured tree, defying in part the need for self-optimisation; and, on the other hand, resource starvation would prevent the structure from growing enough to investigate the structure (re)generation capabilities.

Spawn and destroy policies

Depending on the specific experiment, we may use either a no-spawn/no-destroy policy or one that allows for structural growth and shrinkage. In the latter case, our destroy policy is based on stability and resource threshold: a node leaves the structure if the current amount of resource being received is below a selected threshold and if there are no changes in the amount of resource and success perceived by its neighbours for a certain amount of time (hence, the situation is considered "stable").

The spawn policy is similar, but, in addition, it considers its surroundings: when the amount of resource is above a certain threshold M_ρ, the number of children is below a certain threshold χ,[6] and the surrounding structure is not changing for long enough, then the positions of the neighbouring nodes is considered to decide whether—and where—to spawn a new node.

To decide *where* to spawn a new node, a probabilistic strategy is exploited, aimed at keeping the structure growth as uniform as possible (hence, putting more emphasis on the capability of the system to self-organise the distribution of resources), and yet avoid the creation of an exceedingly large number of nodes too close in space. The rule works as follows.

1. The space around the node is divided into sectors delimited by the segments connecting the current node and each neighbour.
2. All sectors smaller than $2\pi/\chi$ radians are considered occupied and discarded.
3. Among the remaining sectors, one is selected randomly based on the amplitude of the sector (larger arcs have a higher probability of being selected).
4. Once the arc is selected, a new node is spawned at a selected distance from the current node at an angle selected with a Gaussian distribution centred in the centre of the selected sector and with a standard deviation of π/χ radians.
5. If there is no neighbour, the spawn angle is selected uniformly at random.

For instance, if the selected sector spans from angle θ_1 to angle θ_2, the Gaussian distribution will have $\mu = \frac{\theta_2 - \theta_1}{2}$ and $\sigma = \pi/\chi$. Figure 1 visually depicts the angle selection process.

As a special rule, if a node is isolated, it can spawn regardless of the amount of resources it perceives. The special rule is in place to allow "germination" from single nodes (seeds) located in unfortunate positions.

Means of evaluation

Each experiment consists of a parametric simulation template that is simulated multiple times with different parameters. In each simulation template, the structure under study is either created from scratch (via either FieldVMC or VMC), then

[5] Artefact available at https://github.com/angelacorte/fieldVMC

[6] In experiments **(A)** and **(B)**, $\chi = 3$. In experiments of type **(C)**, χ is varied between 2 and 5.

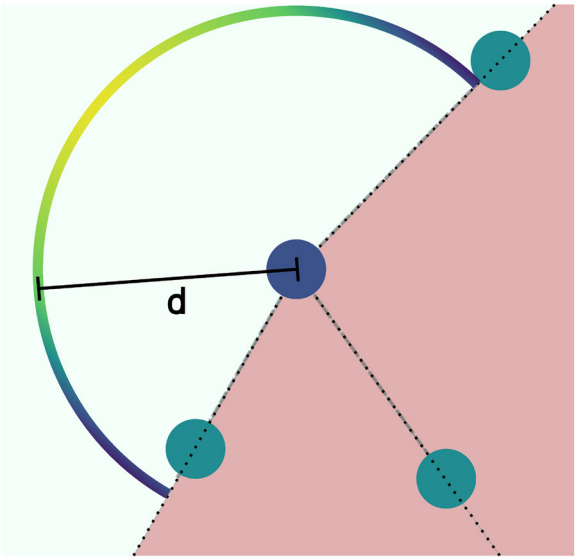

Fig. 1 Spawn angle selection. Discarded sectors are coloured in red. within the selected sector (white) an angle is chosen using a Gaussian distribution centred in the centre of the sector, depicted as a solid line with a color gradient in the Viridis scale (https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html), with ligher colors indicating higher probability. Once the angle is selected, a new node is spawned at a distance d from the current node

possibly perturbed to evaluate some self-organisation capability of FieldVMC.

Such evaluation, in particular, is carried on by leveraging both a qualitative (visual) analysis of the structure evolution, and a quantitative (numerical) analysis of the structure's properties over time.

Qualitative analysis (visualisation)

For the sake of qualitative analysis, we produce graphical representations of pivotal events occurring in the simulations under study. Figures 4, 8, 6 and 10 are examples of such visualisations. In each figure, the cyan area represents the resource and the yellow area the success, with darker shades indicating higher values. Nodes are represented as circles. The root is identified by a dark outer circumference. The size of a circle depends on the amount of resource and success received relative to all other nodes in the system: we fix the maximum possible size D, we compute the maximum amount of resource R and the maximum amount of success S across all nodes in the system; then, for each node in the system with success s and resource r, we determine its size d proportionally to D as $d = \frac{D(r+s)}{R+S}$. Their colour depends on the amount of resource nodes have and is assigned based on the hue of the HSV colour space, with the most resource associated with indigo, and the lowest with red. Dashed lines are communication channels, solid black lines represent the tree structure, and green (resp. orange) lines depict the resource (resp. success) distribution flows, the thicker they are, the more resource (resp. success) is being transferred.

Quantitative analysis (metrics)

To analyse our simulations quantitatively, we define a set of metrics which can be computed w.r.t. each simulation trace, to inspect the status of the structure(s) under study. Which particular metrics are relevant depends on the specific experiment, yet, in general, we are interested in the following metrics:

number of roots: the number of roots in the structure (quantifying the number of disjoint sub-structures which exist at a given moment);

network diameter (of the structure): the length of the maximum shortest path between any two nodes in a structure (quantifying the size of the structure);

number of nodes: the total number of nodes which are part of a structure (quantifying the size of the structure);

node density (of the structure): the number of nodes divided by the total area of the structure's minimal bounding box (quantifying the spatial concentration of nodes within the available space);

degree of the nodes: the average number of neighbours of each node in the structure (quantifying the connectivity of the network);

centroid (of the structure) the average position of all nodes in the structure (synthesising the spatial distribution of nodes);

stabilisation time: the amount of simulated time required by the structure to stabilise, i.e., reaching a state where one or more of the metrics above do not change any more;

data rate: the amount of data exchanged by each node in the system per time unit. This metric is computed by instrumenting the simulation code to serialise each message using Kryo,[7] with no compression.
Additional overhead introduced by aggregate programming is considered by adding 32 bytes for each data item of every message, emulating a SHA-256-based value-tree lookup.

Setup summary

Table 3 summarises the experiments presented in the next sections, detailing the metrics of interest and how parameters are selected.

[7] https://github.com/EsotericSoftware/kryo.

Table 3 List of experiments and relative metrics used in the experiments

Experiment	Metrics	Independent variables
A.1: Self-construction 4.3.2	Stabilisation time	Parameters for VMC are selected arbitrarily, and the corresponding parameters for FieldVMC are found using the optimisation procedure detailed in Eq. 4.3.1
A.2: Self-healing 4.3.3	Stabilisation time, node count	Same as A.1, plus the cut position
B.1: Self-integration 4.4.1	Root count	Initial node count
B.2: Self-division 4.4.2	Same as B.1	Same as B.1
B.3: Self-optimisation 4.4.3	Node count	Same as B.1
C: Self-Optimization with leader election 4.5	Data rate ($^B/_s$)	Available resources ($\max(R)$), and maximum allowed children χ

Experiments of type (A): FieldVMC vs. VMC

Experiments of this sort aim at comparing the performance of FieldVMC with the original VMC model. To the best of our knowledge, there is no agreed-upon definition of performance for VMC. Thus, we compare the two models in terms of *stabilisation time* in a self-construction scenario, and in terms of *adaptability* in a self-healing scenario, which are, in our experience, the two most relevant aspects of VMC which have been studied in the literature so far.

It is worth highlighting that, despite our analysis is conceptually as simple as comparing the two models w.r.t. stabilisation time and adaptability, implementing a *fair* comparison comes with several challenges, namely: the scarce availability of performance data, as well as the lack of an open source reference implementation for VMC. Accordingly, before delving into the details of our experiments, in Sect. "Making the comparison fair", we discuss how to address these challenges and make the comparison as fair as possible.

Making the comparison fair

We provide our own implementation of VMC, which can be run as an Alchemist simulation. Aside from being as close as possible to the original formulation (cf. Sect. "The vascular morphogenesis controller") our implementation can be executed in the same simulation playground as FieldVMC, thus ensuring that the two models are compared in the same conditions.

Baseline VMC implementation details
We implement the original VMC model to in such a way that the update of the vascular pathways (cf. Eq. (3) in Sect. "The vascular morphogenesis controller") is synchronous—coherently w.r.t. the original formulation. To do so, our implementation computes the success (cf. Eq. 1) from the leaves to the root. When the root has received the information from all its children, it computes the amount of resources to send to the children, weighted by the success perceived by the incoming vessels (cf. Eq. (2)). Once the leaves have obtained the resources from the root, the process to evaluate which node can spawn new nodes starts, by propagating the identifier of the node with the highest amount of resources up to the root, so that root can propagate 'spawning signal' back to the leaf node which should spawn.

Overall, our VMC implementation differs from FieldVMC in aspects mostly related to the way they manage the root and resource distribution. In VMC, the root is single and fixed, and there is no mechanism to change it, whereas, in FieldVMC, the root is elected *dynamically*, and there could be multiple roots if the structure is partitioned. Hence, for the sake of fairness, we force FieldVMC to have exactly one fixed root in our comparison experiments, and we use the same root for all the experiments.

Concerning resource management, being FieldVMC dynamic and asynchronous, its root(s) cannot receive a fixed amount of resource at iteration of the algorithm as VMC does, as there is no well-defined concept of complete iteration in FieldVMC. Instead, at each computation round, the root just perceives the amount of resource available in the environment. This creates a discrepancy in the amount of resource available to the root (and thus to the whole structure), which requires dedicated parameter tuning. Similar considerations apply to the amount of success available in the environment.

Parameter tuning
Both VMC and FieldVMC depend on the amount of resources and success available in the environment, and are parametric in the threshold of resources needed to spawn a new node. Under the assumptions we leveraged in Sect. "FieldVMC is a monotonic extension of VMC", these three parameters could be set to the same values in both models; however, in this experiment, those assumptions are not satisfied, as we allow for node spawning and destruction, and we do not enforce synchronous rounds. Since resource and success acquisition is different by construction in the two models (as one is synchronous and the other is asyn-

chronous), these three parameters (the amount of resource and success belonging to the environment, and the spawn threshold of the algorithm) need to be tuned to ensure a fair comparison. We thus perform parameter tuning by setting up an optimisation process aimed at finding the parameter values that minimize the difference between the two models' behaviours.

More precisely, we use VMC as a reference (selecting default parameters values from the literature), we let the algorithm stabilise, and we compute all the metrics defined in Sect. "Quantitative analysis (metrics)". We then adjust the parameters of FieldVMC in such a way that the algorithm a final structure as similar as possible to the one produced by VMC, leveraging on the Nelder–Mead method [50] to minimize the difference of the metrics. Precisely, the parameters being optimised are: (i) the maximum amount of resources, (ii) the maximum amount of success, and (iii) the lower bound of resources needed to spawn a new node.

Formally speaking, we use Nelder–Mead method to minimize the following objective function:

$$\sqrt[n]{\prod_{i=1}^{n}(1+\|\mathsf{m}_i(\text{VMC})-\mathsf{m}_i(\text{FieldVMC})\|)} \tag{4}$$

namely, the geometric mean of the absolute differences between the metrics of the two models (incremented by one to avoid zero values), where n is the number of metrics to equalise, and $\mathsf{m}_i(\ldots)$ is the i-th metric computed on the model in input. The optimisation process proceeds by iteratively updating a simplex, replacing its worst vertex with a new point, allowing it to adapt to the function's landscape and move towards the minimum. The process proceeds until the difference between the values of the objective function at each vertex of the simplex is less than a threshold (which we set to 10^{-6}). Each step of in this optimisation process involves simulating both models multiple times with different random seeds, to ensure the robustness of the results. As per all the experiments presented in this paper, we set the number of simulations to 500 for each parameter combination.

Just for the experiments of type **(A)**, it is only present the Gaussian source of success, and it is placed at (0, 25) within the environment, while the maximum amount of resource is given directly to the root.

Experiment A.1: self-construction

The self-construction experiment measures the ability of the system to autonomously construct a stable structure from scratch.

Goal

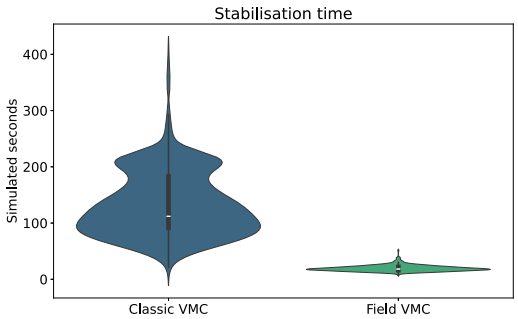

Fig. 2 Results of experiment A.1. Comparing the distribution of the stabilisation time for VMC (blue) and FieldVMC (green) highlights much faster convergence for the latter

The objective is to compare VMC and FieldVMC in terms of convergence time (i.e., the time required for the structure to stabilise) starting from a single node and allowing the system to expand.

Setup We start the simulation with a single node placed at the centre of the resource source. As the simulation begins, the structure naturally expands toward the region offering the highest success, provided that available resources are sufficient to support continued growth. Growth ceases once resource availability becomes inadequate, at which point the structure is expected to stabilise. We define stabilisation as the point at which no new nodes are created or removed for a sustained period (in our case, 90 simulated seconds for three consecutive evaluations). We measure the stabilisation time, which is a proxy for algorithmic efficiency: faster convergence to a stable configuration indicates better performance.

Results Figure 2 presents the results of this experiment. The plot shows the distribution of stabilisation times for VMC (blue) and FieldVMC (green), measured as the time elapsed from the start of the simulation until the structure remains stable according to the criteria defined in Sect. "Quantitative analysis (metrics)". As shown, FieldVMC consistently achieves stabilisation in significantly less time compared to VMC, indicating faster convergence. Figure 3 illustrates the resulting self-constructed structures for both VMC and FieldVMC.

Experiment A.2: self-healing

This experiment investigates the ability of the system to regenerate a stable structure after a disruptive event. We expect the regeneration process to proceed toward areas of higher perceived success, provided sufficient resources are available. Our hypothesis is that the regenerated structure will resemble the original, both in shape and in number of nodes.

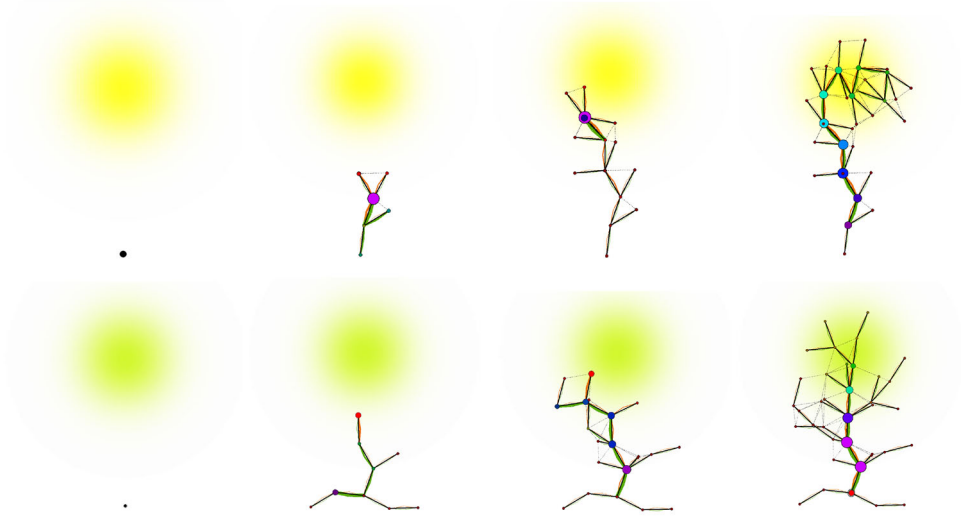

Fig. 3 Self-construction using the classic VMC (top) and FieldVMC (bottom), sequence of simulation snapshots of a randomly selected run. Instructions to read the images are provided in Sect. "Qualitative analysis (visualisation)". From a single node placed in the bottom of the environment (left), we let the system reach stabilisation (right)

Goal The objective is to compare VMC and FieldVMC in terms of adaptability and recovery time after a disruption. More precisely, we observe the system node count in time, thus getting insights on the stabilisation dynamics and the final shape of the structure.

Setup

The simulation starts with a single node and allows the structure to grow naturally. At a fixed point in time ($t = 500$ simulated seconds), we remove a portion of the structure by instantly destroying all nodes above a certain horizontal line at height y. In particular, we cut at $y_A = 24$, $y_B = 18$, $y_C = 9$, and $y_D = 0$, as illustrated in Fig. 4. The system is then allowed to recover and regrow until it reaches stabilisation. To evaluate performance, we measure the node count over time, which should exhibit three phases: initial growth to saturation, a sharp drop at the cut, and recovery back to saturation. We consider a structure stabilised when no node additions or deletions occur for a sustained period, as detailed in Sect. "Quantitative analysis (metrics)".

Results

Results are summarised in Fig. 5. Both algorithms are capable to grow a structure from a single node, and to regenerate it after a disruptive event. However, FieldVMC is faster both in the initial growth phase and in the regeneration phase, converging to a stable configuration more quickly than VMC (coherently with the results of Sect. "Experiment A.1: self-construction"). Interestingly, data shows the implicitly synchronous nature of VMC: for large cuts (y_C and y_D) and

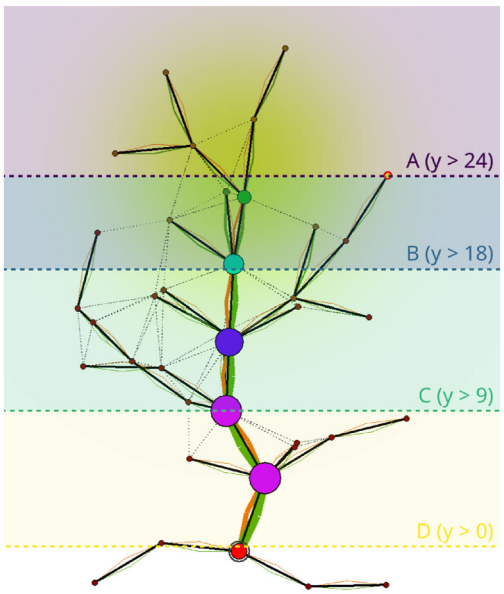

Fig. 4 Cutting strategy used in experiment A.2. We perform a cut by removing all nodes located above a specified horizontal line, defined by a threshold value on the y-axis. The figure illustrates the four cuts, corresponding to thresholds at $y_A = 24$, $y_B = 18$, $y_C = 9$, and $y_D = 0$, respectively

for the initial growth phase, data shows a step-like behaviour, due to the back-and-forth data flow between the root and the leaves. This behaviour is progressively smoothed out as the cut gets smaller. Also interestingly, FieldVMC, despite the

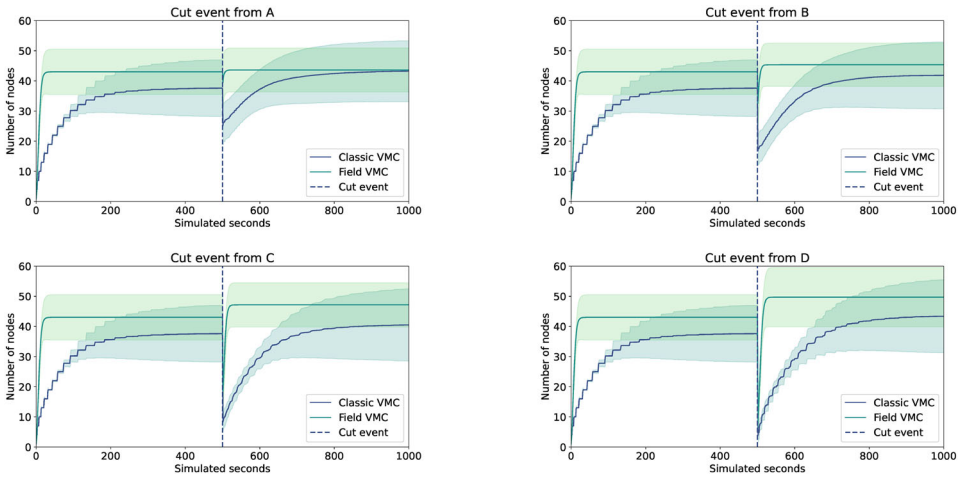

Fig. 5 Results for experiment A.2. We show one plot for each of the cuts depicted in Figure Eq. (4). The vertical dashed line in the middle represents the cutting event, separating the initial growth phase from the regeneration. FieldVMC (green line) faster both in the initial stabilisation (as expected from the results in Sect. "Experiment A.1: self-construction") and regeneration. Coloured shades mark $\pm 1\sigma$ (one standard deviation)

parameter alignment performed in Sect. "Making the comparison fair", tends to stabilise to a larger number of nodes than VMC, both initially and after the regeneration. Both approaches tend to produce slightly larger structures after the regeneration than before the cut, with the effect being more pronounced for larger cuts.

Experiments of type (B): analysing FieldVMC

This set of experiments aims to demonstrate the self-* capabilities of FieldVMC that extend beyond the self-construction and self-healing behaviours already exhibited by VMC. Specifically, we evaluate three advanced properties:

1. *self-integration*, the ability to unify previously separate structures into a single coherent entity;
2. *self-division*, the ability to autonomously split into substructures under stress, without spawning or node destruction; and
3. *self-optimisation*, the ability to evolve the structure toward greater optimality regardless of the initial configuration.

Experiment B.1: self-integration

Goal

This experiment aims to demonstrate the model's ability to integrate multiple structures into a single coherent system, even without spawning or destroying nodes.

Setup

Throughout the experiment, node spawning and destruction are disabled. We initialise the system with two spatially separated structures and allow each to stabilise independently. At $t = 200$ seconds, one structure is instantaneously repositioned in such a way that the two merge. We track the number of roots before and after integration and measure the time required for the system to adapt and stabilise as a unified whole. To evaluate scalability, we repeat the experiment using different node counts per substructure: $N_0 \in \{200, 600, 1000\}$.

Results

Figure 7 summarises the results. Prior to the integration event, the two substructures stabilise independently, each with its own root. After merging, a short transitional phase occurs, during which multiple nodes compete for leadership. Eventually, the system converges into a unified structure with a single root, demonstrating effective self-integration even at increasing scales.

Experiment B.2: self-division

Self-division is the dual of self-integration: it refers to the system's ability to autonomously reconfigure into multiple coherent substructures following a disruptive partitioning event, without spawning or destroying nodes.

Goal

This experiment investigates the system's ability to autonomously divide into two independent structures in response to a network partitioning event. We aim to observe whether the system converges to a stable state with two distinct roots, and to examine the transition dynamics following the disruption.

Setup

We begin with a single, unified structure and allow it to reach stabilisation. At $t = 200$ s, the central portion of the structure is removed, creating two disjoint substructures. We then observe the evolution of the root count and allow the system to stabilise once more. To assess scalability, the experiment is repeated with varying node counts per substructure: $N_0 \in 200, 600, 1000$.

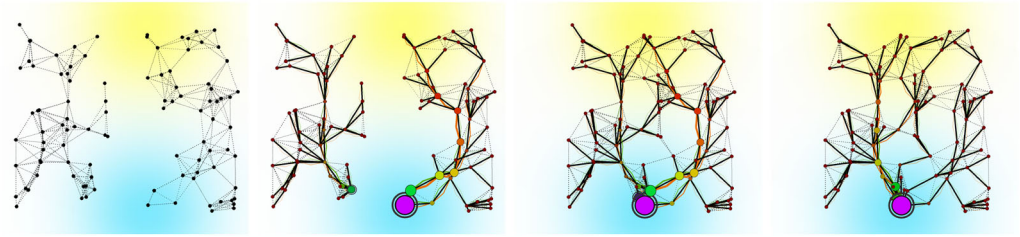

Fig. 6 Grafting (self-integration). Instructions to read the images are provided in Sect. "Qualitative analysis (visualisation)". From a random displacement of nodes divided in two clusters (left) we let the system reach stabilisation (centre-left). Once stabilised, the left cluster is moved toward the right one (centre-right), triggering self-integration (right): the two subsystems merge into a single tree structure, potentially under a new leader, with re-established resource and success flows

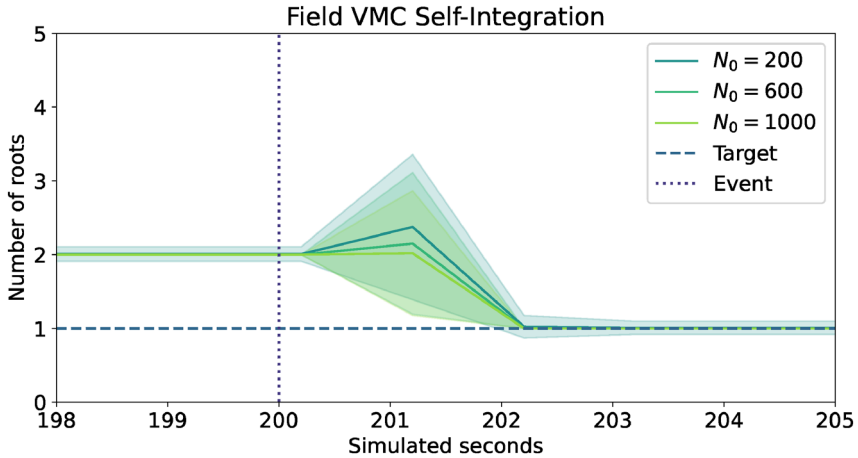

Fig. 7 Data for experiment B.1, where two structures are merged into a single one. The integration event is marked by a dotted line, while the target number of roots is indicated by the dashed line. Shaded areas represent $\pm 1\sigma$ (one standard deviation). Initially, the system stabilises with two independent roots-one per structure. Following the integration event, the system undergoes a brief reorganisation phase, after which it converges to a single integrated structure with one root

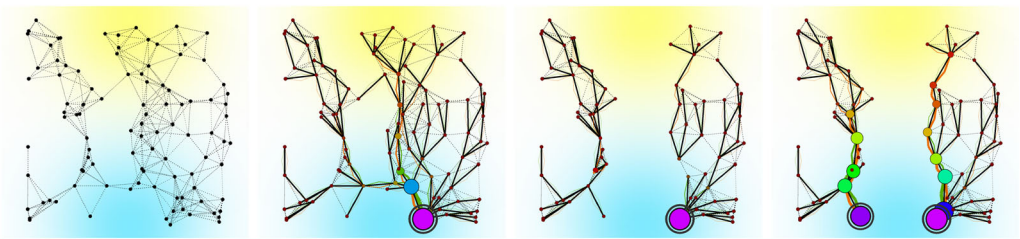

Fig. 8 Cutting (self-division). Instructions to read the images are provided in Sect. "Qualitative analysis (visualisation)". Starting from a random initial distribution of nodes (left), the system is allowed to stabilise into a coherent structure (centre-left). Once stabilised, the central part of the structure is removed (centre-right), creating two disjoint components. The system then autonomously reconfigures itself (right), forming two independent tree structures, each with a leader, and establishing new resource and success flows

Results

Figure 9 summarises the outcomes of the self-division experiment. Prior to the partition, the system stabilises with a single root. After the disruption, it eventually converges to two stable substructures, each with its own root. However, in contrast to the rapid convergence observed in the self-integration experiment (Fig. 7), self-division is markedly slower and more sensitive to system scale. This discrepancy stems from the implementation of the data-distribution mechanism in FieldVMC. Information is propagated using a simple adaptive variant of the Bellman–Ford algorithm, which is known to suffer from the *slow rising value* problem [26, 51]. In this phenomenon, the propagation rate of information is bounded by the shortest neighbour-to-neighbour

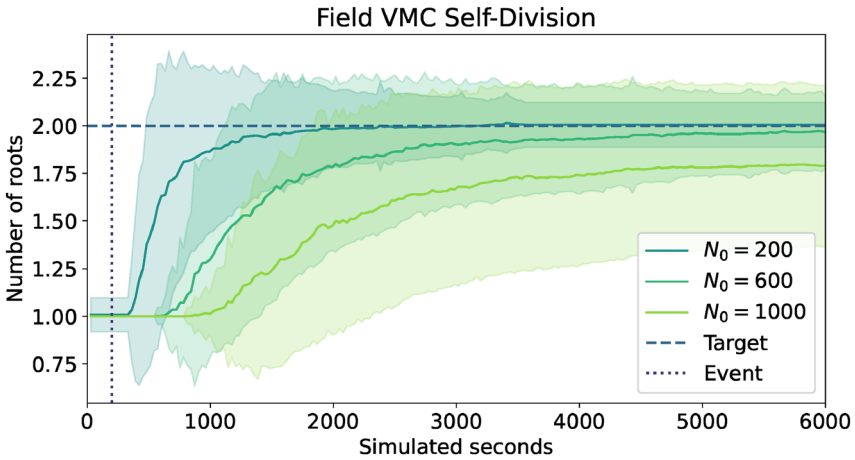

Fig. 9 Results for self-division (Sect. "Experiment B.2: self-division"), where a single structure is split into two disjoint components. The division event is marked by a dotted line, and the target number of roots (i.e., two) by the dashed line. Shaded areas represent $\pm 1\sigma$ (one standard deviation). Initially, the system maintains a single root; following the disruption, it transitions to a state with two roots as the structures stabilise independently. In contrast to the almost instantaneous nature of self-integration (Fig. 7), self-division occurs on a much longer timescale. Furthermore, self-division is strongly affected by system scale: as the number of nodes increases, the time required for reorganisation grows significantly

distance in the network. As a result, denser networks, where such distances are statistically smaller, exhibit significantly slower convergence, revealing a performance limitation in the current implementation.

Experiment B.3: self-optimisation

Self-optimisation is the capability of the system to converge to a stable, resource-efficient structure, regardless of the initial configuration.

Goal The goal is to show the ability of the model to optimise the structure to be resource-efficient, building a structure that does not waste resources on non-successful areas.

Setup
For self-optimisation, both spawning and destruction of nodes are enabled. The system starts with an initial configuration where $N_0 \in \{1, 10, 100, 300, 500, 1000\}$ nodes are randomly distributed in the environment in the area surrounding the resource and the success sources, but not overlapped or close to either of them. We let the evolve until stabilisation, measuring the overall node count. We expect the system to converge to a similar structure regardless of the initial configuration: although the shape of the structure may vary, the number of nodes should stabilise around a similar value.

Results
Results are summarised in Fig. 11. Regardless of the initial number of nodes, the system converges to a similar structure, counting about 40 nodes. Interestingly, starting from very few nodes (e.g., $N_0 = 1$) or from many nodes (e.g., $N_0 = 1000$) leads to larger variance in the final structure. This effect is due to the larger number of spawning and destruction events required to reach the final structure, which introduces more randomness in the process.

Experiments of type (C): resource scaling analysis

The experiment presented in this section aims at evaluating how FieldVMC scales in terms of network resource consumption under varying network densities and scales. More precisely, we measure the data rate and analyse (i) the average data rate for each device, and (ii) the system's total data rate.

Setup
We replicate the conditions of the self-optimisation scenario (cf. Sect. "Experiment B.3: self-optimisation") with $N_0 = 1$, as it is includes leader election, node spawning, and destruction. With respect to the self-optimisation experiment, we add two additional free variables: (i) the maximum amount of resources available in the environment $\max(R) \in \{100, 250, 500, 1000\}$, and (ii) the maximum count of children per node $\chi \in \{2, 3, 4, 5\}$. This experimental design enables a systematic evaluation of how varying network densities and scales influence the data rate requirements of FieldVMC. Specifically, configurations with limited resources will yield networks with fewer nodes, while configurations with a lower maximum number of spawnable children will result in both smaller and sparser networks. Since the number of messages to be sent per node is proportional to the number of neighbours, we expect that the mean data rate per node will increase with the network density. Also, we expect that the overall data rate of the system will

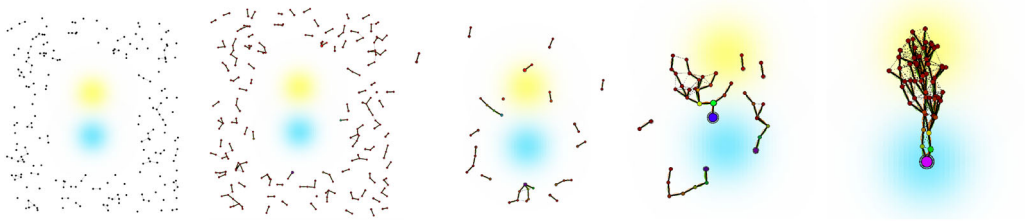

Fig. 10 Self-optimisation (Sect. "Experiment B.3: self-optimisation"). Instructions to read the images are provided in Sect. "Qualitative analysis (visualisation)". Nodes ($N_0 = 200$ in these snapshots) are initially deployed randomly in an area surrounding the resource and success sources. With time, multiple structures emerge, progressively converging towards the resource source and optimising the node count as they encounter and merge. The final structure (rightmost image) is resource-efficient, with the root close to the resource source, and several "leaves" extracting success

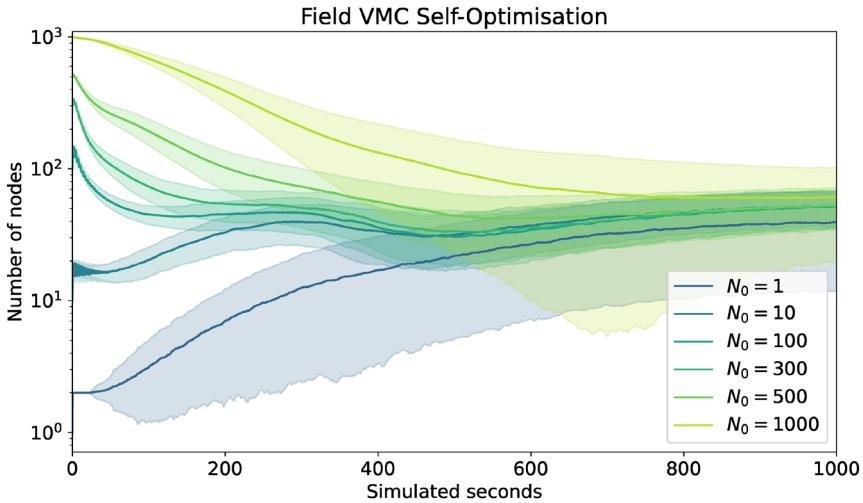

Fig. 11 Results for self-optimisation (Sect. "Experiment B.3: self-optimisation") for different initial node count (N_0). Shaded areas represent $\pm 1\sigma$ (one standard deviation). Regardless of the initial number of nodes, the system converges to similar structures with about 40 nodes. Starting with very few nodes (e.g., $N_0 = 1$), or with many nodes (e.g., $N_0 = 1000$), leads to larger variance, as more destructions and spawn events are needed to reach the final structure, and each of them introduces a source of randomness

increase sharply with $\max(R)$, and more gently with χ (as both parameters influence the final network size).

Results The results of this experiment are summarised in Figs. 12 and 13. Results highlight that both the average and overall data rates increase with higher network densities and scales, but in any case the data rate stabilises as the system configuration becomes increasingly stable. We observe that the impact of χ is relatively limited, especially when $\max(R)$ is low: the available resources are the bottleneck limiting the network size, and thus the structure tends to grow more in depth, elongating towards the success source, rather than in breadth. This is evident by observing how increasing χ has almost no effect on the behaviour of the system with low resources, but increases the data rate requirements noticeably when more resources are available.

Looking at raw numbers, devices require between about 1 $^{KB}/_{s}$ and 10 $^{KB}/_{s}$ to sustain the protocol, depending on the network density, assuming rounds to be executed at 1 Hz on average. A centralised architecture (for instance an MQTT-based one) with devices operating as the same frequency, would require the server to be capable of handling between about 6 $^{KB}/_{s}$ and 400 $^{KB}/_{s}$, depending primarily by scale, even though density can have a measurable influence. This is a reasonable requirement for most IoT devices; additionally, lowering the average operation frequency linearly reduces the data rate (but, of course, it also linearly increases the convergence time of the system). More efficient data exchange formats (such as protocol buffers), compression, and shorter hashes could further reduce this requirement. If compression or shorter hashes are used, however, two relevant trade-offs must be considered: compression comes at the expense of increased computational load (and thus power drain), while using shorter hashes for the aggregate protocol increases the collision risk, potentially impacting correctness.

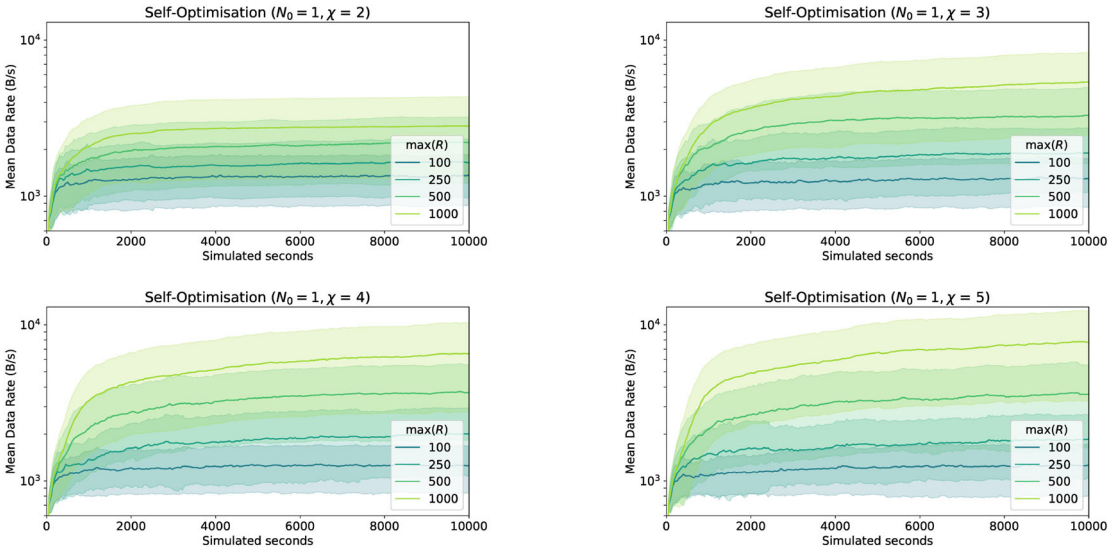

Fig. 12 Results for experiment C, described in Sect. "Experiments of type **(C)**: resource scaling analysis". We show one plot for each configuration of spawnable children χ, for different amounts of available resources $\max(R_r^l)$. Coloured shades mark $\pm 1\sigma$ (one standard deviation). The system starts with a single node, and the data rate is initially low, as there are few nodes and neighbours to communicate with. As the network density and scale increase, the average data rate per node also increases, due to the higher number of neighbours each node must communicate with. However, as the system stabilises, the data rate tends to stabilise as well

Limitations and threats to validity

Scope and assumptions. The approach models the transport, split, and consumption of a *single conserved scalar* ("resource") over a local-communication graph. It assumes reliable neighbourhood discovery at the chosen interaction radius, non-Byzantine participants, and eventual delivery under benign asynchrony (fair scheduling). Mapping the abstract resource to a domain variable (e.g., energy, bandwidth, workload) requires calibration and possibly domain-specific safeguards.

Performance and scalability. Each device exchanges constant-size state with its neighbours per round; local work scales with node degree, and extra state is limited to neighbourhood buffers. Under the tree/synchronous restriction, stabilisation of success and resource requires at most twice the hop-diameter. On arbitrary graphs with churn, convergence/recovery time increases with the effective diameter and the rate of topology change; tight worst-case bounds under adversarial dynamics are not provided.

Robustness to asynchrony and churn. Field-based execution tolerates moderate message delay and reordering, but extreme churn can induce temporary oscillations (e.g., parent flapping) and slower settling. Hysteresis and backoff mitigate oscillations yet introduce latency trade-offs. We do not address malicious or arbitrarily faulty nodes; resilience here is to benign failures and transient disconnects.

Sensitivity and parametrisation. Behaviour depends on neighbourhood radius, update cadence, weighting functions, and lifecycle thresholds for growth/pruning. Mis-tuning can yield slow diffusion ("slow rising values" [51]) and delayed division/merging. Adaptive or learned parameter selection and adoption of "fast-healing" information diffusion algorithms [52] is left as future work.

Optimality and guarantees. The controller is decentralised and myopic by design. It *does not* guarantee global optimality of routing/assignment or minimal message cost; it aims for robust, scalable behaviour with local information. Formal guarantees beyond the tree/synchronous compatibility result (e.g., bounds for general dynamic graphs, multi-leader stability, competitive ratios) remain open.

Simulation realism and deployment. Results are in-silico. Reality gaps may arise due to sensing noise, radio asymmetry, clock drift, mobility, and platform constraints. Deployment requires engineering of message budgets, safety guards (e.g., capacity caps, watchdogs), and a domain-specific mapping from the abstract resource to physical quantities.

Security and adversarial settings. We do not model adversarial inputs or Byzantine behaviour. Integrating trust filters [53], outlier rejection [54], and signed neighbourhood state is future work for safety-critical deployments.

External validity. The reported scenarios and topologies, while varied, cannot cover all regimes (e.g., extreme sparsity/density, highly heterogeneous delays). Generalisation to

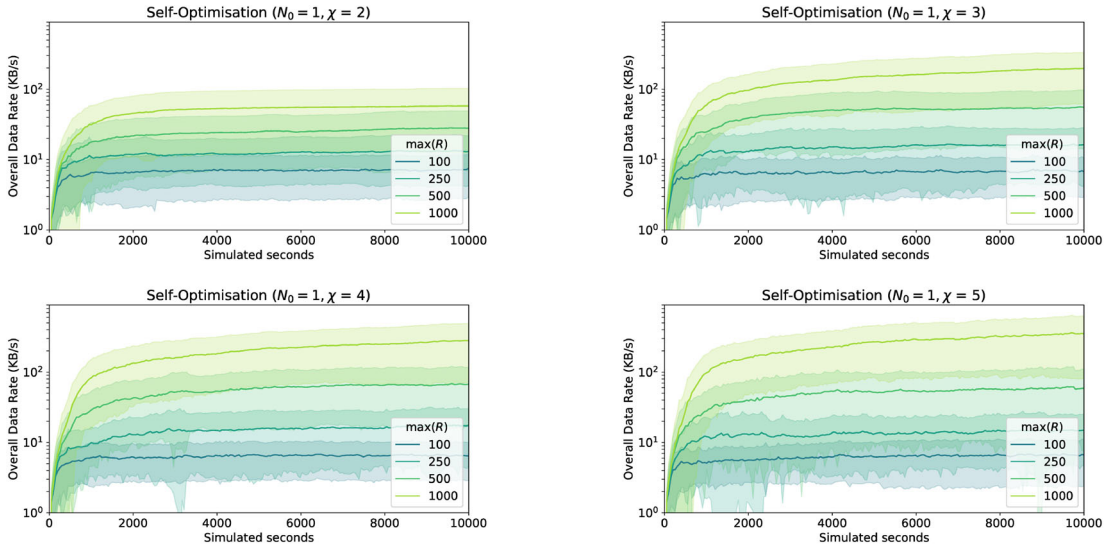

Fig. 13 Results for experiment C, described in Sect. "Experiments of type **(C)**: resource scaling analysis". We show one plot for each configuration of spawnable children χ, for different amounts of available resources $\max(R_r^t)$. Coloured shades mark $\pm 1\sigma$ (one standard deviation). The system starts with a single node, and the overall data rate is initially low, as there are few nodes and neighbours to communicate with. As the network density and scale increase, the overall data rate also increases, due to the higher number of nodes and neighbours in the system. However, as the system stabilises, the overall data rate tends to stabilise as well

markedly different environments should be considered with care and re-evaluated empirically.

Related work

This paper covers FieldVMC, a field- or aggregate computing-based generalisation of VMC, which is an approach for regulating the growth of artificial structures in a self-organising way, finding applications in networks and robotics as mentioned in Sect. "Introduction". This problem, as well as similar problems, also known under terms like morphogenesis [2, 7], formation control [55], self-organisation [40], self-assembly [7, 15], self-reconfiguration [56], etc. can be addressed through multiple paradigms. We provide an overview of such paradigms, which are often applicable to artificial collectives [57] or distributed systems in general, and then focus on the two ingredients of FieldVMC: VMC and aggregate computing.

Paradigms for morphogenesis and related applications

Morphogenetic engineering approaches can be classified in different taxonomies. For instance, in [58], approaches are distinguished according to the type of dynamical process sustaining the morphogenetic process: (i) *construction*, where few agents assemble together or attach blocks, (ii) *coalescence*, where the agents of a swarm move to form shapes, (iii) *growth*, where division or aggregation mechanisms are used to form biological-like organisms, and (iv) *generation*, where grammar-like rewrite rules are applied to progressively transform a system. In the following, we classify the morphogenetic approaches based on their underlying paradigm.

Optimisation approaches could in principle be used to find the best morphologies, but due to the complexity arising from the mixed and multi-objective nature of the problem, heuristic or learning approaches are generally preferred [59].

Learning-based approaches use machine learning techniques such as reinforcement learning, evolutionary algorithms, or a combination of these [60] to generate emergent behaviour. For the VMC model, evolutionary algorithms are used to evolve parameters of the model [2]. Though generally suffering from state explosion, machine learning techniques can be powerful and interesting, but are quite different from the "manual" approach to controller design followed in this paper (though attempts to combine aggregate computing with machine learning have been done and still subject to research [61]).

Behaviour-based approaches [62] break down complex collective behaviour into a hierarchy of reactive behavioural blocks. Each behaviour reacts to input and provides outputs, and arbitration mechanisms are used to determine what behaviours are active at a certain time. Liu and Winfield [63] have proposed a behaviour-based decentralised morphology control mechanism for communicating SYMBRION robots endowed with infrared sensing.

Agent-based approaches [64] model each system component as an agent, an autonomous entity that interacts with its environment as well as other agents, e.g., by exchanging messages or using the environment as a medium (cf. stigmergy). For morphogenesis, these models emphasise proliferation, migration, and differentiation mechanisms [64].

Generative approaches [65] use production rules to grow structures. A notable example is L-system [66], a grammar-based parallel rewriting system where generated strings are translated into geometric structures, then extended, e.g., by so-called *swarm grammars* [67]. Another example is growing point language (GPL) [68], based on "tropisms" that, based on simulated chemical signals, attract or repel the motion of growing points.

Physics models are also used. A notable class is that of *reaction-diffusion approaches*, which can be used to generate repetitive patterns by using inhibitors and activators to regulate the concentration of entities in space [64].

Macroprogramming approaches [25] abstract from individual nodes and the details of message-passing, and rather provide high-level abstractions, often domain-specific. Aggregate (field-based) computing is an example of macroprogramming: we cover its recent developments in Sect. "Aggregate computing".

Our FieldVMC follows the aggregate computing approach under the VMC morphogenesis model. The different classes of approaches work quite differently and are not directly comparable: a deep comparison is beyond the scope of this paper. To the best of our knowledge, the main attempt adopting aggregate computing for pattern formation is given by the "functional blueprint" approach by Beal [69]: here, shapes are formed by inducing "distortions" on manifolds represented by a multitude of devices. It is related to our approach (composing "aggregate computing" building blocks to achieve collective behaviour), but it uses a different, *geometric* morphogenesis model based on coalescence.

Vascular morphogenetic controller

Since the preliminary works [12] and the introduction of VMC [2], research has been carried out to study how the model can support different kinds of applications.

In [14], mixed societies including both robots and natural plants are considered, with robot controllers evolved in order to regulate the growth of plans by steering their tips. In [15], VMC is used for the self-assembly of robot swarms into tree-like shapes. This is then investigated for *adaptive* self-assembly [16], where light sources are used to induce changes in the environment. In [17], a variation of VMC allowing the root th change over time is used to induced amoeboid-like locomotion. In [19], VMC is used to address collective distribution of resources. In [20], VMC is adopted to regulate the adaptation of business organisations (reframed as VMC trees) operating in dynamic economic environments. In [70], an approach loosely based on VMC is used for *asset distribution* in multi-scale systems.

Aggregate computing

We covered aggregate computing background in Sect. "Aggregate programming and field calculus". More broadly, the aggregate computing thread of research spans multiple areas in computer science, including computational models and formalisms (i.e., field calculi [29, 32]), programming languages and libraries (e.g., Protelis [33], ScaFi [36], FCPP [38]), algorithms (e.g., leader election [42]), abstractions and patterns (e.g., aggregate processes, and self-organisation patterns like SCR [40]), middleware and deployment models (e.g, pulverisation [71]), and applications (e.g., swarm robotics [72]).

Concerning formalisms, the eXchange Calculus [32] is the most recent proposal as well as the most minimal (in terms of language constructs with special semantics) and general: it subsumes earlier higher-order field calculi [29] and provides the original constructs (`nbr`, `rep`, etc.) in terms of a single `exchange` construct. FieldVMC is implemented in *Collektive*, an implementation of the eXchange Calculus, and is therefore up-to-date from the point of view of backing formalisms.

Aggregate computing applications generally span distributed and collective computing scenarios like in the IoT [24, 73], smart cities [74], and swarm robotics [72]. An alternative implementation of FieldVMC could be rooted in MacroSwarm [72], a recent aggregate programming library designed to control robotic swarms, resulting in a different syntactic flavour of FieldVMC. However, MacroSwarm would be most useful to implement other morphogenetic models, such as those based on coalescence (see Sect. "Paradigms for morphogenesis and related applications").

Conclusion

In this work, we proposed FieldVMC, a novel formulation of the vascular morphogenesis controller grounded in the aggregate programming paradigm. By recasting VMC as a field-based computation, we overcame the structural and synchronisation limitations of the original model, enabling asynchronous and decentralised execution over arbitrary topologies. This generalisation preserves the core principles of VMC—namely, the self-organising emergence of hierarchical structures—while enhancing expressiveness, scalability, and applicability.

We provided a full implementation of FieldVMC using a functional Kotlin Multiplatform aggregate programming

framework. By providing FieldVMC as a reusable, open-source artefact, we facilitate its adoption for the design and engineering of decentralised morphogenetic systems.

The model was validated through in-silico experiments that reproduced canonical VMC dynamics and demonstrated novel phenomena such as self-integration, self-division, and self-optimisation. These results show that FieldVMC is a monotonic extension of VMC: it subsumes its original behaviour and expands its range of application to more dynamic and complex scenarios.

Future work

Building on the results presented in this paper, several directions for future investigation emerge.

Convergence dynamics In our experiments, we focused on the final structure produced by FieldVMC and on its convergence time. However, the dynamics of convergence and the intermediate structures produced during the process are also of interest. We plan to investigate the convergence dynamics of FieldVMC in more detail, including the intermediate structures produced during the convergence process, and how they relate to the final structure. A deeper analysis of these could reveal underlying self-organisation patterns, potentially leading to the identification of reusable morphogenetic building blocks. Different compositions of such blocks could, in turn, yield alternative morphogenetic algorithms.

Faster convergence In this work, we focused on the correctness of the model and on overcoming the limitations of VMC. Although FieldVMC converges significantly faster than VMC in our experiments, we did not explore optimisations to further accelerate convergence. Indeed, we used stand-alone approaches to the construction of upstream and downstream flows, using strategies derived from the simplest implementations of self-healing gradients and convergecasts. However, it is well-known that these approaches suffer from slow convergence in some conditions [51], and that more sophisticated approaches can significantly improve convergence speed [46]. We plan to investigate the application of these techniques to FieldVMC in future work to achieve faster convergence.

Real-world validation While VMC has been successfully applied to robotics [15], FieldVMC has so far only been validated in-silico. We aim to validate FieldVMC beyond simulation, by targeting deployment on physical platforms such as modular or swarm robotic systems. Given the implementation in Kotlin Multiplatform and the underlying aggregate computing model, porting to embedded or distributed cyber-physical architectures is technically feasible and would offer insights into practical applicability.

Non-biological applications Although biologically inspired, FieldVMC is not confined to plant metaphors. We envision applying it to non-biological domains such as organisational modelling, distributed computing, and complex system design. In particular, FieldVMC may serve as a decentralised model for self-organising teams, responsive hierarchies, or resilient resource management. We plan to explore these applications in future work, starting with organisational scenarios where FieldVMC can model self-organising teams or adaptive workflows.

Towards a morphogenetic library Just as gradient-based coordination gave rise to libraries of reusable self-organising behaviours [26, 47], FieldVMC could mark the beginning of a library of morphogenetic reusable blocks. Composition and reuse of these blocks could enable the design of complex morphogenesis-inspired systems, allowing for the creation of sophisticated structures and behaviours with no centralised control, by combining simpler morphogenetic primitives.

Author Contributions Angela Cortecchia: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data Curation, Writing-original draft, Writing-Review and Editing, Visualization. Giovanni Ciatto: Methodology, Validation, Formal analysis, Resources, Writing-original draft, Writing-Review and Editing, Visualization. Roberto Casadei: Conceptualization, Writing-original draft, Writing-Review and Editing, Visualization. Danilo Pianini: Conceptualization, Methodology, Software, Validation, Formal analysis, Resources, Writing-original draft, Writing-Review and Editing, Visualization, Supervision, Project administration.

Funding This work was partially supported by the Italian Ministry of University and Research (MUR) under the National Recovery and Resilience Plan (PNRR) through the project FAIR-Future Artificial Intelligence Research, funded by the European Union-NextGenerationEU, with the following details: PNRR–M4C2–Investment 1.3, Partenariato Esteso PE00000013–"FAIR-Future Artificial Intelligence Research"–Spoke 8 "Pervasive AI".

Data Availability The data can be obtained by reproducing the experiments as described in the available code repository; due to strict reproducibility policies, the resulting data will be exactly the same.

Code Availability The code is available at the public repository https://github.com/angelacorte/fieldVMC and on the Zenodo repository at https://doi.org/10.5281/zenodo.17361508.

Declarations

Conflict of interest All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

Ethical approval and consent to participate This article does not contain any studies with human participants or animals performed by any of the authors.

Consent for publication Not applicable.

Materials availability Not applicable.

References

1. Alattas RJ, Patel S, Sobh TM (2018) Evolutionary modular robotics: survey and analysis. J Intell Robot Syst 95(3–4):815–828. https://doi.org/10.1007/s10846-018-0902-9
2. Zahadat P, Hofstadler DN, Schmickl T (2017) Vascular morphogenesis controller: a generative model for developing morphology of artificial structures. In: Bosman PAN (ed) Proceedings of the genetic and evolutionary computation conference, GECCO 2017, Berlin, July 15–19, 2017. ACM, New York, pp 163–170. https://doi.org/10.1145/3071178.3071247
3. Calvo Garzón P, Keijzer F (2011) Plants: adaptive behavior, root-brains, and minimal cognition. Adapt Behav 19(3):155–171
4. Scherer J, Rinner B (2016) Persistent multi-uav surveillance with energy and communication constraints. In: IEEE international conference on automation science and engineering, CASE 2016, Fort Worth, August 21–25, 2016. IEEE, Piscataway, pp 1225–1230. https://doi.org/10.1109/COASE.2016.7743546
5. Yu K, O'Kane JM, Tokekar P (2019) Coverage of an environment using energy-constrained unmanned aerial vehicles. In: International conference on robotics and automation, ICRA 2019, Montreal, May 20–24, 2019. IEEE, Piscataway, pp 3259–3265. https://doi.org/10.1109/ICRA.2019.8794150
6. Meng Y, Jin Y (2011) Morphogenetic self-reconfiguration of modular robots. In: Meng Y, Jin Y (eds) Bio-inspired self-organizing robotic systems. Studies in computational intelligence, vol 355, pp 143–171. https://doi.org/10.1007/978-3-642-20760-0_7
7. O'Grady R, Christensen AL, Dorigo M (2021) SWARMORPH: morphogenesis with self-assembling robots. In: Doursat R, Sayama H, Michel O (eds) Morphogenetic engineering, toward programmable complex systems. Springer, Berlin, pp 27–60. https://doi.org/10.1007/978-3-642-33902-8_2
8. Chafik AA, Gaber J, Tayane S, Ennaji M, Bourgeois J, El-Ghazawi TA (2024) From conventional to programmable matter systems: a review of design, materials, and technologies. ACM Comput Surv 56(8):210–121026. https://doi.org/10.1145/3653671
9. Mottola L, Picco GP (2011) MUSTER: adaptive energy-aware multisink routing in wireless sensor networks. IEEE Trans Mob Comput 10(12):1694–1709. https://doi.org/10.1109/TMC.2010.250
10. Joshi D, Jha V (2025) Energy-efficient multi-static sink model for wireless sensor network. J Supercomput 81(8):961. https://doi.org/10.1007/S11227-025-07418-0
11. Snyder PL, Greenstadt R, Valetto G (2009) Myconet: a fungi-inspired model for superpeer-based peer-to-peer overlay topologies. In: 3rd IEEE international conference on self-adaptive and self-organizing systems, SASO 2009, San Francisco, September 14–18, 2009. IEEE Computer Society, Piscataway, pp 40–50. https://doi.org/10.1109/SASO.2009.43
12. Wahby M, Hofstadler DN, Heinrich MK, Zahadat P, Hamann H (2016) An evolutionary robotics approach to the control of plant growth and motion: modeling plants and crossing the reality gap. In: 10th IEEE international conference on self-adaptive and self-organizing systems, SASO 2016, Augsburg, September 12–16, 2016. IEEE Computer Society, Piscataway, pp 21–30. https://doi.org/10.1109/SASO.2016.8
13. Mariano P, Salem Z, Mills R, Zahadat P, Correia L, Schmickl T (2017) Design choices for adapting bio-hybrid systems with evolutionary computation. In: Bosman PAN (ed) Genetic and evolutionary computation conference, Berlin, July 15–19, 2017, Companion material proceedings. ACM, New York, pp 211–212. https://doi.org/10.1145/3067695.3076044
14. Hofstadler DN, Wahby M, Heinrich MK, Hamann H, Zahadat P, Ayres P, Schmickl T (2017) Evolved control of natural plants: crossing the reality gap for user-defined steering of growth and motion. ACM Trans Auton Adapt Syst 12(3):15–11524. https://doi.org/10.1145/3124643
15. Soorati MD, Ghofrani J, Zahadat P, Hamann H (2018) Robust and adaptive robot self-assembly based on vascular morphogenesis. In: 2018 IEEE/RSJ international conference on intelligent robots and systems, IROS 2018, Madrid, October 1–5, 2018. IEEE, Piscataway, pp 4282–4287. https://doi.org/10.1109/IROS.2018.8594093
16. Soorati MD, Zahadat P, Ghofrani J, Hamann H (2018) Adaptive path formation in self-assembling robot swarms by tree-like vascular morphogenesis. In: Correll N, Schwager M, Otte MW (eds) Distributed autonomous robotic systems, the 14th international symposium, DARS 2018, Boulder, October 15–17, Springer proceedings in advanced robotics, vol 9. Springer, Berlin, pp 299–311 (2018). https://doi.org/10.1007/978-3-030-05816-6_21
17. Zahadat P, Schmickl T (2018) Locomotion as a result of displacement of resources. In: Ikegami T, Virgo N, Witkowski O, Oka M, Suzuki R, Iizuka H (eds) 2018 conference on artificial life, ALIFE 2018, Tokyo, July 23–27, 2018. MIT Press, Cambridge, pp 232–233. https://doi.org/10.1162/ISAL_A_00048
18. Zahadat P, Hofstadler DN, Schmickl T (2018) Morphogenesis as a collective decision of agents competing for limited resource: A plants approach. In: Dorigo, M., Birattari, M., Blum, C., Christensen, A.L., Reina, A., Trianni, V. (eds.) Swarm Intelligence - 11th International Conference, ANTS 2018, Rome, Italy, October 29-31, Proceedings. Lecture Notes in Computer Science, vol. 11172, pp. 84–96. Springer, Berlin, Heidelberg (2018). https://doi.org/10.1007/978-3-030-00533-7_7
19. Zahadat P, Hofstadler DN (2019) Toward a theory of collective resource distribution: a study of a dynamic morphogenesis controller. Swarm Intell 13(3–4):347–380. https://doi.org/10.1007/S11721-019-00174-X
20. Zahadat P, Diaconescu A (2020) Reactive or stable: a plant-inspired approach for organisation morphogenesis. In: Bongard JC, Lovato JL, Hébert-Dufresne L, Dasari R, Soros LB (eds) 2020 conference on artificial life, ALIFE 2020, Online, July 13–18, 2020. MIT Press, Cambridge, pp 614–622. https://doi.org/10.1162/ISAL_A_00244
21. Viroli M, Beal J, Damiani F, Audrito G, Casadei R, Pianini D (2019) From distributed coordination to field calculus and aggregate computing. J Log Algebra Methods Program 109:100486. https://doi.org/10.1016/j.jlamp.2019.100486
22. Mamei M, Zambonelli F, Leonardi L (2004) Co-fields: a physically inspired approach to motion coordination. IEEE Pervasive Comput 3(2):52–61. https://doi.org/10.1109/MPRV.2004.1316820
23. Lluch-Lafuente A, Loreti M, Montanari U (2017) Asynchronous distributed execution of fixpoint-based computational fields. Log Methods Comput Sci. https://doi.org/10.23638/LMCS-13(1:13)2017

24. Beal J, Pianini D, Viroli M (2015) Aggregate programming for the internet of things. IEEE Comput 48(9):22–30. https://doi.org/10.1109/MC.2015.261
25. Casadei R (2023) Macroprogramming: concepts, state of the art, and opportunities of macroscopic behaviour modelling. ACM Comput Surv 55(13s):275–127537. https://doi.org/10.1145/3579353
26. Viroli M, Audrito G, Beal J, Damiani F, Pianini D (2018) Engineering resilient collective adaptive systems by self-stabilisation. ACM Trans Model Comput Simul 28(2):1–28. https://doi.org/10.1145/3177774
27. "angelacorte/fieldvmc: 1.34.8," Oct. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17361508
28. Cortecchia A, Pianini D, Ciatto G, Casadei R (2024) An Aggregate Vascular Morphogenesis Controller for Engineered Self-Organising Spatial Structures.In: IEEE international conference on autonomic computing and self-organizing systems, ACSOS 2024, Aarhus, Denmark, September 16-20, 2024. IEEE, pp 133–138. https://doi.org/10.1109/ACSOS61780.2024.00032
29. Audrito G, Viroli M, Damiani F, Pianini D, Beal J (2019) A higher-order calculus of computational fields. ACM Trans Comput Log 20(1):1–55. https://doi.org/10.1145/3285956
30. Mamei M, Zambonelli F (2009) Programming pervasive and mobile computing applications: the TOTA approach. ACM Trans Softw Eng Methodol 18(4):1–56. https://doi.org/10.1145/1538942.1538945
31. Casadei R, Dente F, Aguzzi G, Pianini D, Viroli M (2023) Self-organisation programming: a functional reactive macro approach. In: IEEE international conference on autonomic computing and self-organizing systems, ACSOS 2023, Toronto, September 25–29, 2023. IEEE, Piscataway, pp 87–96. https://doi.org/10.1109/ACSOS58161.2023.00026
32. Audrito G, Casadei R, Damiani F, Salvaneschi G, Viroli M (2024) The exchange calculus (XC): a functional programming language design for distributed collective systems. J Syst Softw 210:111976. https://doi.org/10.1016/J.JSS.2024.111976
33. Pianini D, Viroli M, Beal J (2015) Protelis: practical aggregate programming. In: Proceedings of the 30th annual ACM symposium on applied computing, Salamanca, April 13–17, 2015, pp 1846–1853. https://doi.org/10.1145/2695664.2695913
34. Voelter M, Benz S, Dietrich C, Engelmann B, Helander M, Kats LCL, Visser E, Wachsmuth G (2013) DSL engineering—designing, implementing and using domain-specific languages. http://www.dslbook.org
35. Bettini L (2016) Implementing domain-specific languages with Xtext and Xtend, 2E. Packt Publishing, Birmingham
36. Casadei R, Viroli M, Aguzzi G, Pianini D (2022) Scafi: a scala dsl and toolkit for aggregate programming. SoftwareX 20:101248. https://doi.org/10.1016/j.softx.2022.101248
37. Riti P (2017) Internal DSL. Apress, pp 45–57. https://doi.org/10.1007/978-1-4842-3036-7_3
38. Audrito G, Torta G (2024) FCPP to aggregate them all. Sci Comput Program 231:103026. https://doi.org/10.1016/J.SCICO.2023.103026
39. Sedunov A (2022) Kotlin in-depth: a guide to a multipurpose programming language for server-side, front-end, android, and multiplatform mobile. BPB Publications. https://books.google.it/books?id=hrhoEAAAQBAJ
40. Pianini D, Casadei R, Viroli M, Natali A (2021) Partitioned integration and coordination via the self-organising coordination regions pattern. Future Gener Comput Syst 114:44–68. https://doi.org/10.1016/j.future.2020.07.032
41. Wolf TD, Holvoet T (2007) 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, July 9–11, 2007. IEEE Computer Society, Piscataway, pp 295–298. https://doi.org/10.1109/SASO.2007.16
42. Pianini D, Casadei R, Viroli M (2022) Self-stabilising priority-based multi-leader election and network partitioning. In: IEEE international conference on autonomic computing and self-organizing systems, ACSOS 2022, Virtual, September 19–23, 2022. IEEE, Piscataway, pp 81–90. https://doi.org/10.1109/ACSOS55765.2022.00026
43. Mo Y, Audrito G, Dasgupta S, Beal J (2022) Near-optimal knowledge-free resilient leader election. Automatica 146:110583. https://doi.org/10.1016/J.AUTOMATICA.2022.110583
44. Dijkstra EW (1974) Self-stabilizing systems in spite of distributed control. Commun ACM 17(11):643–644. https://doi.org/10.1145/361179.361202
45. Nagpal R, Shrobe HE, Bachrach J (2003) Organizing a global coordinate system from local information on an ad hoc sensor network. In: IPSN, LNCS, vol 2634. Springer, Berlin, pp 333–348. https://doi.org/10.1007/3-540-36978-3_22
46. Audrito G, Casadei R, Damiani F, Viroli M (2017) Compositional blocks for optimal self-healing gradients. In: 11th IEEE international conference on self-adaptive and self-organizing systems, SASO 2017, Tucson, September 18–22, 2017. IEEE Computer Society, Piscataway, pp 91–100. https://doi.org/10.1109/SASO.2017.18
47. Fernandez-Marquez JL, Serugendo GDM, Montagna S, Viroli M, Arcos JL (2013) Description and composition of bio-inspired design patterns: a complete overview. Nat Comput Int J 12(1):43–67. https://doi.org/10.1007/S11047-012-9324-Y
48. Audrito G, Casadei R, Damiani F, Pianini D, Viroli M (2021) Optimal resilient distributed data collection in mobile edge environments. Comput Electr Eng 96:107580. https://doi.org/10.1016/j.compeleceng.2021.107580
49. Pianini D, Montagna S, Viroli M (2013) Chemical-oriented simulation of computational systems with ALCHEMIST. J Simul 7(3):202–215. https://doi.org/10.1057/jos.2012.27
50. Nelder JA, Mead R (1965) A simplex method for function minimization. Comput J 7(4):308–313. https://doi.org/10.1093/COMJNL/7.4.308
51. Beal J, Bachrach J, Vickery D, Tobenkin MM (2008) Fast self-healing gradients. In: Wainwright RL, Haddad H (eds) Proceedings of the 2008 ACM symposium on applied computing (SAC), Fortaleza, March 16–20, 2008. ACM, New York, pp 1969–1975. https://doi.org/10.1145/1363686.1364163
52. Audrito G, Damiani F, Viroli M (2018) Optimal single-path information propagation in gradient-based algorithms. Sci Comput Program 166:146–166. https://doi.org/10.1016/J.SCICO.2018.06.002
53. Casadei R, Aldini A, Viroli M (2018) Towards attack-resistant aggregate computing using trust mechanisms. Sci Comput Program 167:114–137. https://doi.org/10.1016/J.SCICO.2018.07.006
54. Pianini D, Ciatto G, Casadei R, Mariani S, Viroli M, Omicini A (2018) Transparent protection of aggregate computations from byzantine behaviours via blockchain. In: Furini M, Mirri S, Bouchard K, Guidi B (eds) Proceedings of the 4th EAI international conference on smart objects and technologies for social good, GOODTECHS 2018, Bologna, November 28–30, 2018. ACM, New York, pp 271–276. https://doi.org/10.1145/3284869.3284870
55. Li W, Wang Z, Guan C, Chen C, Wang B, Ren P, Qiu Y, Zhang Q, Wang H, Wang D, Zhuang J, Xu B, Hao Z, Fan Z (2024) Formation control of swarm robots: a survey from biological inspirations to design automation methods. https://doi.org/10.2139/ssrn.4941922
56. Dai Y, He S, Nie X, Rui X, Li S, He S (2024) Research on reconfiguration strategies for self-reconfiguring modular robots: a review. J Intell Robot Syst. https://doi.org/10.1007/s10846-024-02067-6

57. Casadei R (2023) Artificial collective intelligence engineering: a survey of concepts and perspectives. Artif Life 29(4):433–467. https://doi.org/10.1162/ARTL_A_00408
58. Doursat R, Sayama H, Michel O (2013) A review of morphogenetic engineering. Nat Comput 12(4):517–535. https://doi.org/10.1007/s11047-013-9398-1
59. Li W, Wang Z, Mai R, Ren P, Zhang Q, Zhou Y, Xu N, Zhuang J, Xin B, Gao L, Hao Z, Fan Z (2023) Modular design automation of the morphologies, controllers, and vision systems for intelligent robots: a survey. Vis Intell 1(1):2. https://doi.org/10.1007/s44267-023-00006-x
60. Whitelam S, Tamblyn I (2019) Learning to grow: control of materials self-assembly using evolutionary reinforcement learning. arXiv:1912.08333 [CoRR abs]
61. Aguzzi G, Casadei R, Viroli M (2022) Machine learning for aggregate computing: a research roadmap. In: 42nd IEEE international conference on distributed computing systems, ICDCS Workshops, Bologna, July 10, 2022. IEEE, Piscataway, pp 119–124. https://doi.org/10.1109/ICDCSW56584.2022.00032
62. Balch TR, Arkin RC (1998) Behavior-based formation control for multirobot teams. IEEE Trans Robot Autom 14(6):926–939. https://doi.org/10.1109/70.736776
63. Liu W, Winfield AFT (2010) Autonomous morphogenesis in self-assembling robots using IR-based sensing and local communications. Springer, Berlin, pp 107–118. https://doi.org/10.1007/978-3-642-15461-4_10
64. Glen CM, Kemp ML, Voit EO (2019) Agent-based modeling of morphogenetic systems: advantages and challenges. PLoS Comput Biol 15(3):1006577. https://doi.org/10.1371/journal.pcbi.1006577
65. Stillman NR, Mayor R (2023) Generative models of morphogenesis in developmental biology. Semin Cell Dev Biol 147:83–90. https://doi.org/10.1016/j.semcdb.2023.02.001
66. Lindenmayer A (1975) Developmental algorithms for multicellular organisms: a survey of l-systems. J Theor Biol 54(1):3–22. https://doi.org/10.1016/s0022-5193(75)80051-8
67. Mammen S, Jacob C (2009) The evolution of swarm grammars—growing trees, crafting art, and bottom-up design. IEEE Comput Intell Mag 4(3):10–19. https://doi.org/10.1109/MCI.2009.933096
68. Morgan A, Coore D (2013) Extending the growing point language to self-organise patterns in three dimensions. In: Blum C, Alba E (eds) Genetic and evolutionary computation conference, GECCO '13, Amsterdam, July 6–10, 2013, Companion material proceedings. ACM, New York, pp 109–110. https://doi.org/10.1145/2464576.2466799
69. Beal J (2011) Functional blueprints: an approach to modularity in grown systems. Swarm Intell 5(3–4):257–281. https://doi.org/10.1007/s11721-011-0056-x
70. Zahadat P, Diaconescu A (2022) The impact of multi-scale control topology on asset distribution in dynamic environments. In: 2022 IEEE international conference on autonomic computing and self-organizing systems companion (ACSOS-C). IEEE, Piscataway, pp 31–36. https://doi.org/10.1109/acsosc56246.2022.00023
71. Farabegoli N, Pianini D, Casadei R, Viroli M (2024) Scalability through pulverisation: declarative deployment reconfiguration at runtime. Future Gener Comput Syst 161:545–558. https://doi.org/10.1016/J.FUTURE.2024.07.042
72. Aguzzi G, Casadei R, Viroli M (2025) Macroswarm: a field-based compositional framework for swarm programming. Log Methods Comput Sci. https://doi.org/10.46298/LMCS-21(3:13)2025
73. Casadei R (2025) System-wide IoT design and programming: patterns for decentralised collective processes. Internet Things 29:101436. https://doi.org/10.1016/J.IOT.2024.101436
74. Aguzzi G, Casadei R, Pianini D, Viroli M (2022) Dynamic decentralization domains for the internet of things. IEEE Internet Comput 26(6):16–23. https://doi.org/10.1109/MIC.2022.3216753