Macro-to-micro behavioural mappings in distributed systems: a characterisation on event structures

1st Roberto Casadei
Dept. of Computer Science and Engineering
Alma Mater Studiorum—Università di Bologna
Cesena, Italy
0000-0001-9149-949X
roby.casadei@unibo.it

2nd Paolo Baldini
Dept. of Computer Science and Engineering
Alma Mater Studiorum—Università di Bologna
Cesena, Italy
0000-0002-3625-5414

3rd Niccolò Castronuovo
Liceo "A. Einstein"
Rimini, Italy
0000-0002-1054-0161

***Abstract*—Engineering the self-organising behaviour of artificial collective systems, such as robot swarms, is challenging. Effective abstractions, provided by programming paradigms tailored to the purpose, can simplify the task of designers by promoting conceptual and practical tools supporting reasoning and implementation. Macro-programming has emerged as an umbrella term for paradigms (like aggregate and choreographic programming) aimed at specifying collective behaviours by a global perspective, largely abstracting micro-level details. In this work, we characterise the macro-programming setting by leveraging the formal framework of event structures, which represents collective computations as causal graphs of interactions among computation events. Specifically, we identify different types of macro-to-micro mechanisms, namely ways of mapping macroscopic specifications into local policies, allowing to conceptualise notable paradigms like aggregate and choreographic programming as specific instances of macro-programming.**

***Index Terms*—macro-programming, self-organisation, collective computation, programming paradigms, distributed artificial intelligence, swarms**

I. Introduction

Artificial collective systems are organisations of devices that interact between each other and operate in group [4]. The computation is notably distributed across the devices and often lacks a predefined centralised control point. This provides some advantages with respect to centralised systems, such as enhanced resiliency to failures and scalability to the number of devices and computational load [13]; conversely, however, this makes distributed systems more difficult to design and control.

Over the years, the research community investigated the potential of various approaches to simplify the design and control of such systems. Among the most prominent, we can cite automatic design and learning [8], [10], and programming languages devised specifically for the control of multiple agents [3], [12]. The former two use training data and simulations to create solutions that maximise specific metrics (e.g., effectiveness); the latter focus on abstracting the problem from the programming of the single entities of the collective to that of the entire system. Specifically, in this work, we will consider this latter approach, which we refer to as *macro-programming* [6].

This contribution is supported by the Italian Science Fund (FIS3) Starting Grant project FoMaSE – Foundations for Macro-programming-based Software Engineering (Grant No. FIS-2024-00174, CUP J53C25002170001).

While using macro-programming approaches, programmers focus on describing the behaviour of the entire system rather than that of its constituent entities. This high-level and general description can be then instantiated into the code of the specific entities involved by means of some sort of compilation [14] and/or projection [11], or it can be executed directly by the individual devices [5], [12]. The result is that the programmer abstracts away from low-level implementation and synchronisation details, focusing only on the high-level logic of the computation.

While macro-programming of collective adaptive systems have shown expressiveness and effectiveness in defining the group behaviour, the current landscape of technical implementations is characterised by a multitude of paradigms that differ under many aspects. Therefore, with this work, we aim to characterise some aspect of macro-programming so as to unify different approaches under a common framework. We do so by exploiting the framework of *augmented event structures* as used e.g. in [1]. This offers a way of representing a collective computation. On top of this framework, we provide a computational interpretation of the events occurring on the devices, formalised as a macro-to-micro function mapping devices to local time-dependent control policies. We begin by describing background and related works in Section II. We then characterise macroscopic programming in Section III and propose some examples in Section IV. Finally, we discuss the characterisation we propose in Section V and draw the conclusions of the work in Section VI.

II. Background and Related Work

In this section, we describe notable background work in the context of macro-programming. We initially describe augmented event structures, that are the abstract computation model on which we develop our work. Following, we present aggregate computing and choreographic programming, two paradigms to macro-program large groups of devices that we use as recurring examples throughout the manuscript. Finally, we discuss previous work characterising space-time interpretations of local computation functions.

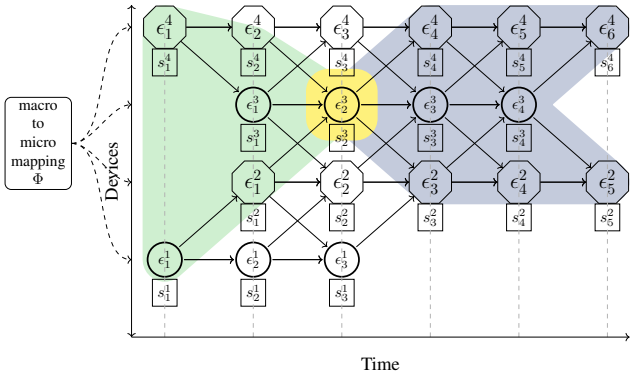

Fig. 1: Example of a collective dynamics modelled as an event structure. Nodes labelled by ϵ_k^δ denote the k-th sense–compute–interact round of device δ. Given a reference event (yellow), its causal past (green) and future (grey) are determined by the messaging relation (i.e., the arrows). The figure also shows an important addition of this work: the *macro-to-micro localisation function* Φ that compiles a macro-program into policies of the single devices.

A. Augmented event structures

Augmented event structures, as used for instance in [1], are representations of executions of situated collective computations, which are based on causal chains of interconnected events (where an event denotes a computation step or "round") over time. See Figure 1 as an example. They are formally defined as a 4-tuple $\mathbf{E} = \langle E, \rightsquigarrow, d, s \rangle$ where:

- E is a countable set of events,
- $\rightsquigarrow \subseteq E \times E$ is a relation between events,
- d: $E \to \Delta$ is a mapping from events to the devices where they occurred,
- s: $E \to S$ is a mapping from events to (some representation of) sensors status.

We also denote with $N(\epsilon)$ the neighbours of ϵ, i.e., the set of events $\{\epsilon' \in E \mid \epsilon' \rightsquigarrow \epsilon\}$. Additionally, we identify for any device $\delta \in \Delta$ the set of events $E_\delta = \{\epsilon \in E \mid d(\epsilon) = \delta\}$ that occur on that device. Those form a chain, meaning that there are no distinct events $\epsilon, \epsilon_1, \epsilon_2 \in E_\delta$ such that either $\epsilon \rightsquigarrow \epsilon_i$ for $i = 1, 2$ or $\epsilon_i \rightsquigarrow \epsilon$ for $i = 1, 2$. Furthermore, the transitive closure of $\rightsquigarrow$ forms a strict partial order $< \subseteq E \times E$, called *causality relation*. Finally, the set of events $X_\epsilon = \{\epsilon' \in E \mid \epsilon' < \epsilon\} \cup \{\epsilon' \in E \mid \epsilon \rightsquigarrow \epsilon'\}$ is finite for all ϵ (i.e., $\rightsquigarrow$ and $<$ are locally finite).

B. Aggregate computing and programming

Aggregate computing is a macro-programming paradigm devised to define the behaviour of a group of embodied agents [13]. It leverages *field abstractions* to perform operations on the whole system, therefore abstracting away from local computation. Specifically, functions of an aggregate program can take and return fields. This is the case of the `gradient(s)` function, that takes a Boolean field of sources and returns the field of corresponding distance estimates by running a shortest-path computing algorithm. The only requirement of aggregate computing is the presumption of some operative conditions, such as computational synchrony, connectivity, and computing power.

A peculiar characteristic of aggregate computing is that every device remains an autonomous entity, locally executing the same program using information collected from neighbours. Specifically, each device perceives the environment through its sensors and gathers the most recent messages asynchronously received from its neighbours. This information, which constitutes the agent *local context*, is used by the aggregate program during computation to determine intended operation and communication. Communication is strictly local, meaning that a node can interact only with its immediate neighbours (e.g., based on proximity). Notably, communication failures and changes in the set of connected devices will be gradually absorbed, permitting the system fields to eventually stabilise.

Examples provided in the following pages use ScaFi [5], a Scala-internal aggregate programming language implementation.

C. Choreographic programming

Choreographic programming [11] is a development paradigm for distributed and concurrent systems, where programs express *protocols*, namely interactions between processes (a.k.a. *parties* or *components*), by a *global* perspective. The paradigm leverages mechanisms (cf. *endpoint projection*) to transform the choreographic program into the executable software of each component. Notably, this takes care of low-level implementation details, such as managing communication, permitting the programmer to focus on the macroscopic behaviour of the distributed system.

Although several choreographic programming languages have been proposed, they share a number of common features [11]. Communication primitives constitute their core building blocks. A communication instruction typically specifies an expression to be evaluated by one process together with a variable of another process where the resulting value is stored. The transmitted expression may consist of (i) a constant, (ii) a variable, or (iii) a function application. Most languages also provide primitives for updating local variables, enabling stateful computations. Consequently, a choreographic program is naturally expressed as a sequence of communications interleaved with local computations that progressively evolve the state of the participating processes.

D. Space-time function interpretations

In the effort to characterise macro-programming paradigms, we cannot avoid to talk about the article of Audrito et al. [2], which attempts to classify aggregate computing functions according to their **S**pace-**T**ime dependency (i.e., **I**ndependence, **C**ontinuous-dependence, **D**iscrete-dependence). These classes are organised according to their generality, specifically: *independent* < *continuously-dependent* < *discretely-dependent*. In this classification:

- a SI function output is independent of communication with neighbours;

- a TI function is self-stabilising;
- a SI–TI function is local (e.g., mathematical functions);
- SI-TD, SD-TI, SD-TD functions typically compute discrete properties of networks of devices (e.g., device counts, leader election);
- SI-TC, SC-TI, SC-TC functions typically compute space-time (geometrical) properties (e.g., time and spatial distances).

While Audrito et al. [2] consider the space-time interpretation of local computation functions, we focus on the mapping from the global specification of the macro-program to the policies of devices, unifying different paradigms as specific instances of macro-programming. This perspective implicitly encompasses also paradigms whose characteristics could not be satisfactorily captured by the classification of Audrito et al., such as choreographies, that can consider also devices following distinct policies.

III. Macroscopic Programming: a Characterisation

In this section, we use the formal framework of event structures to characterise different ways to map a single (global) specification of distributed behaviour (from which we abstract) into the local computations carried out by the devices of a collective. This will serve to provide a conceptual characterisation of *macro-to-micro* mechanisms that may be implemented by concrete macro-programming languages.

A. Macro-to-micro localisation

Definition 1 (Macro-to-micro localisation process): Let Δ be the set of devices. We define a *macro-to-micro localisation process* as a function:

$$\Phi : \Delta \to \mathcal{M}$$

mapping each device $\delta \in \Delta$ to a (possibly higher-order) time-dependent policy

$$\mu_t^\delta := \Phi(\delta) \in \mathcal{M},$$

where $\mathcal{M}$ is a family of local computational functions originated (through projection, compilation, or replication) from a single global program (i.e., the macro-program) defining the *collective* behaviour of the group.

Each policy μ_t^δ determines the computational behaviour of the device $d(\epsilon)$ during the event ϵ occurring at time t; thus, we can identify the occurrence of an event ϵ on device δ such that $d(\epsilon) = \delta$ at time t, with the execution of the policy μ_t^δ on the local context available at that event. Such computation is treated abstractly as a policy that maps local information available at an event to an output value, without committing to any specific algorithmic realisation.

Definition 2 (Time slice): A time slice $E_t \subseteq E$ is a set of events representing a consistent global observation, i.e., a cut of the causality relation.

A time slice $E_t \subseteq E$ provides the correspondence between the global notion of time and the event structure. In particular, E_t collects all events that belong to the same global observation instant t. Therefore, when the macro-to-micro localisation process Φ associates a time-dependent policy μ_t^δ to a device δ, the parameter t is interpreted through the time slice E_t: all events $\epsilon \in E_t$ occurring on device δ execute the computational behaviour specified by μ_t^δ.

Definition 3 (Compatibility conditions): The macro-to-micro localisation process Φ is admissible only if it satisfies the following consistency requirements. We assume a universe of *capabilities* $\mathcal{C}$ and a universe of *requirements* $\mathcal{R}$. Capabilities represent properties or resources that a device may provide, such as sensors, actuators, communication interfaces, computational resources, or security mechanisms. Requirements represent the resources or services that a policy needs in order to be meaningfully instantiated. We model capabilities and requirements of devices through functions:

$$cap : \Delta \to \mathcal{P}(\mathcal{C}), \qquad req : \mathcal{M} \to \mathcal{P}(\mathcal{R})$$

where $cap(\delta)$ denotes the set of capabilities available on device δ, and $req(\mu)$ denotes the requirements declared by the policy μ. A compatibility relation:

$$\models \subseteq \mathcal{P}(\mathcal{C}) \times \mathcal{P}(\mathcal{R})$$

specifies when a set of capabilities satisfies a set of requirements. In the simplest case,

$$cap(\delta) \models req(\mu) \quad \iff \quad req(\mu) \subseteq cap(\delta),$$

although more sophisticated notions of compatibility may be considered. The macro-to-micro localisation process Φ must then satisfy the following conditions. For every device $\delta \in \Delta$, the policy μ assigned by Φ can be instantiated on δ only if it satisfy a *capability condition*, i.e., the device satisfies the requirements declared by the policy:

$$\forall \delta \in \Delta, \qquad cap(\delta) \models req(\Phi(\delta)).$$

This condition guarantees that computational behaviours are deployed only on devices that provide the resources needed for their execution. A similar notion of compatibility based on matching device capabilities against component requirements is employed in [9].

Definition 4 (Local information space): If $\delta \in \Delta$ is any device and if $\epsilon \in E$ with $d(\epsilon) = \delta$ is an event occurring at time t, the policy

$$\mu_t^\delta : \mathcal{I}_\epsilon \longrightarrow \mathcal{O}_\epsilon,$$

is a mapping from the space of causes (a.k.a., inputs) of the event ϵ to the space of effects (a.k.a., outputs) of the same event. The set $\mathcal{I}_\epsilon$ comprises messages received by other devices, local sensor observations, and the device state, basically representing the local context of the device. On the other hand, the set $\mathcal{O}_\epsilon$ includes messages to be sent to other devices and operations to be performed on the device state and on the environment.

B. Macro-programming regimes as restrictions of the macro-to-micro localisation event semantics

We show that standard variants of macro-programming can be uniformly expressed within the macro-to-micro localisation process Φ and the augmented event structure E introduced above. Recall that Definition 1 associates each device $\delta \in \Delta$ with a policy μ_t^δ. We now characterise different macro-programming regimes as constraints on the macro-to-micro localisation process function Φ.

Definition 5 (Static macro-programming): Static macro-programming is a form of macro-programming in which each device is associated with a fixed policy that is used unchanged across all computation rounds. Formally, given any device δ and any pair of distinct instants of time t_1 and t_2, we have

$$\mu_{t_1}^\delta = \mu_{t_2}^\delta.$$

Hence, the macro-to-micro localisation process depends only on the device identity and is invariant over time.

Definition 6 (Dynamic macro-programming): Dynamic macro-programming is a form of macro-programming in which the policy associated with a device is self-adaptive, i.e., it may change from round to round.

Thus, in dynamic macro-programming, there exists at least one device δ whose associated policy changes with time across execution. In particular, each event may compute using a different local rule depending on its position in the causal evolution of the device. We remark that the dependence on time of the policy μ_t^δ can be either self-regulating (i.e., the policy can modify itself from round to round) or externally controlled (e.g., by a global clock or by an external controller).

Definition 7 (Uniform macro-programming): Uniform macro-programming is a form of macro-programming in which all devices locally use the same policy:

$$\mu_t \in \mathcal{M}$$

such that:

$$\forall \delta \in \Delta, \qquad \Phi(\delta) = \mu_t.$$

If μ_t is also invariant over time, then we have a special case of uniform static macro-programming, where the same policy is executed by all devices at all times. In this regime, the computational semantics is completely homogeneous across both devices and time, and all heterogeneity arises solely from the event structure $(E, \rightsquigarrow, <)$ and the local inputs $\mathcal{I}_\epsilon$. Still, some sort of *computational* heterogeneity is considered, for example when the replicated policy considers some behavioural branching according to the local context of the device (i.e., it behaves differently in different situations). *Aggregate computing models* typically fall into this category, where the same program is executed by all devices at all times.

Definition 8 (Heterogeneous macro-programming): In heterogeneous macro-programming different devices use different policies.

Unlike uniform macro-programming, no constraint enforces equality of the policies across devices, thus the mapping Φ has at least two distinct policies in its image. Even in the heterogeneous case, the policy of each device may still be invariant over time, leading to a static heterogeneous macro-programming regime. Notice that if none of the above constraints are imposed, we obtain a fully general heterogeneous macro-programming regime where each event can have an arbitrary policy, leading to maximal expressiveness but minimal structure.

Unifying view: All considered macro-programming paradigms can be expressed as different structural constraints on the macro-to-micro localisation process $\Phi : \Delta \to \mathcal{M}$. In particular:

- *static* macro-programming constrains Φ to give each device a time-independent policy;
- *dynamic* macro-programming allows full dependence on time;
- *uniform* macro-programming constrains Φ to be constant across space (namely across computational loci);
- *heterogeneous* macro-programming constrains Φ not to be constant across space.

Hence, macro-programming regimes can be understood as a hierarchy of restrictions on the expressive power of the computational assignment function Φ, independently of the underlying event structure semantics.

C. Elements of macro-programming

Definition 9 (Macro-input): The macroscopic input for the system at time T_{start} is given by the sensor states $\{s^*_{T_{\text{start}}}\}$ and the initial topology $\epsilon^*_{T_{\text{start}}} \rightsquigarrow \epsilon^*_{T_{\text{start}}+1}$.

The macroscopic input of a computation is the information available on the events belonging to an initial time slice $E_{T_{\text{start}}}$. Specifically, it is determined by sensor states associated with the events in $E_{T_{\text{start}}}$ and the communication topology induced by the neighbourhood relation $\rightsquigarrow$ restricted to such events.

Definition 10 (Macro-output): The macroscopic output for the system at time T_{end} is given by a result and possibly a side effect:

- *macro-result*: (a function of) the output of events $\epsilon^*_{T_{\text{end}}}$;
- *macro-effect*: (a function of) the environment state as provided by sensor states $s^*_{T_{\text{end}}}$.

The macro-result is obtained from the values produced by the events $\epsilon \in E_{T_{\text{end}}}$, while the macro-effect is obtained from the environment state resulting from the operations generated by those events.

Definition 11 (Macro-programming goal): The goal of macro-programming is to devise a mapping from events ϵ_i^δ to functions of the event states that enables to obtain the desired macro-input/macro-output mapping for a set of considered event structures.

Given an event structure $\mathbf{E}$ and a macro-to-micro localisation process Φ, the macro-programming problem can be viewed as the synthesis of a mapping that transforms the information available on an initial time slice $E_{T_{\text{start}}}$ into the desired observations on a final time slice $E_{T_{\text{end}}}$. The macro-to-micro localisation process Φ can be interpreted as the restriction of a more general space-time assignment

$$\Theta : X \times T \to \mathcal{M},$$

where X denotes the physical space inhabited by the collective and T the time domain. The map Θ associates a computational behaviour with every point of space and time, thus providing a global description of the intended system dynamics. Thus, given an event ϵ happening in the point (x, t) of the space-time, the value $\Theta(x, t)$ represents the policy of the computation of this events. The localisation process Φ is then recovered by restricting Θ to the actual events of the execution. More precisely, whenever an event ϵ occurs at time t and at position x on device δ, the policy followed at that event is given by:

$$\Theta(x, t) = \Phi(\delta) = \mu_t^\delta.$$

Under this interpretation, macro-programming can be viewed as the problem of defining a global space-time computational law Θ whose restriction to the event structure induces the desired local computations and, consequently, the desired macroscopic behaviour.

IV. Examples

Example 1 (Aggregate computing as uniform macro-programming): As described in Section II-B, aggregate computing permits a single program to specify the global behaviour of a system while remaining executable locally on individual nodes. This characteristic designates this approach as a form of uniform macro-programming. A typical toy example [13] is given by a *gradient field* computation, where every device computes the shortest path length from a possibly dynamic set of *source* devices:

Listing 1: Example of aggregate computing program

```
def distanceTo(s) {
  rep(∞) {
    (dist) => mux(s, 0, minHood(nbr{dist}+nbrRange()))
  }
}
```

Specifically, each non-source member of the collective collects distance information from its neighbours with **`nbr`**, sums its distance to them (computed with `nbrRange`), and selects the minimum with `minHood`. The repetition across space and time of the local computation permits the propagation of the distances across the network, eventually making every agent converge to the correct distance value.

Example 2 (Choreographic programming as heterogeneous macro-programming): As described in Section II-C, choreographic programming pertains the "compilation" of individual components software by projecting (specialising) a human-written program defining the behaviour of the whole system. This characteristic contextualises this approach within the paradigm of heterogeneous macro-programming. Cruz-Filipe et al. [7] propose a simple but significant example by discussing a choreography describing the interaction of two processes, `p` and `r`, that should share information according to a condition known only by `p` (see Listing 2). Specifically, `p` must verify if its value `p.c` is equal to a local value `x`; `p` then communicates to `r` the outcome of the comparison: label `L` in the positive case and R otherwise. Finally, `p` sends to `r` its value `p.c` in case of equality, with `r` sending its value `r.c` to `p` otherwise.

Listing 2: Example of choreographic program

```
C = if p.c == x
    then (p→r[L]; p.c→r; 0)
    else (p→r[R]; r.c→p; 0)
```

This implies that `p` and `r` must behave differently, with the latter evaluating and informing the former of the condition and the former waiting for its reception before potentially initiating operations. The global program describing their interaction is thus projected into distinct software for each of the two processes, rendering the choreography an approach for heterogeneous macro-programming.

V. Discussion

In this work, we propose a conceptual framework that provides a unifying perspective for various paradigms designed to govern the macroscopic behaviour of collective systems, such as choreographic and aggregate programming. Indeed, despite some notable differences, many approaches share enough similarities to be considered part of the same family. Identifying the essential / common characteristics among them could foster the propagation of ideas, the comparison of solutions, and the identification of general concerns and mechanisms that may be further explored. In this light, our work attempts to identify the essential elements characterising macro-programming paradigms so as to capture a large class of possible concrete instantiations. Specifically, we pursue a sufficiently high abstraction level to reconcile divergent design approaches within a single, coherent perspective.

A. Comparison with other characterisations

The aim to unify different macro-programming paradigms according to the event structure abstraction stands out as the main contribution of this work. Indeed, this is what differentiates it from previously proposed characterisations, such as the one introduced by Audrito et al. [2], that investigate the properties and characteristics of specific macro-programming paradigms (e.g., in Audrito et al. the authors focus on aggregate computing only). Therefore, this work proposes a further step in the characterisation of macro-programming, permitting to abstract a larger family of programming paradigms.

B. Macro-to-micro localisation process

The main element that we identify is a macro-to-micro mapping function that, starting from an abstracted macro-program, associates possibly different policies to devices in a collective. This permits to represent the process of deploying / compiling the macro-program on / for specific devices, possibly according to their capabilities, role, and location[1]. This captures, for instance, the process of *projection* employed by choreographies [11], the *code-splitting* used in

[1]The mapping of the macro-program into microcode could enable specifying even advanced operational properties, such as the identification / selection of leaders, the definition of the group topology, etc.

multi-tier programming [14], and the copy of the macro-program to every device in aggregate computing [5] (i.e., the function is a replicator). Furthermore, this process could also identify and separate code to be executed in different time instants, effectively identifying some operational policy that changes over time. Practically, this function acts as a macro-to-micro mapping, instantiating the macro-program into device-executable code (a.k.a. localisation).

C. The event and policy abstractions

Our proposal to represent the computation of events as policies also enables describing the constructs employed by different programming paradigms in a unified manner. For instance, some approaches focus on the messages exchanged between devices, while others consider the higher abstraction of "information updates" (e.g., how the value of the neighbours changed). This is the case of choreographies and aggregate computing, that respectively model the interaction between agents and the continuous update of the collective state. The model that we consider abstracts both messages exchanged by devices and field-state updates as *inputs* to the policy of the agent, permitting to define a coherent behavioural policy framework for diverse paradigms.

D. Unifying communication and state oriented paradigms

While event inputs capture the practical drivers of the computation (i.e., messages, state-updates, etc.), they also permit to assimilate different computational approaches. Specifically, they permit to work with both (i) interaction protocols and (ii) global states (i.e., fields) oriented approaches, including the spectrum of options between them.

We consider a programming paradigm "communication oriented" if the program-logic considers primarily the interaction of the different entities involved in the computation. More precisely, the computation is *communication-determined* if the control-flow primarily determines the outcome of the execution, with the global state playing at most a contextual (non-branching) role. One extreme example of this situation is that of stateless devices that consider the state of the collective indirectly by means of received messages, such as, for instance, in some instances of choreographic models; in this context, the macro-program only defines the interaction protocol between devices with different roles.

Dually, a programming approach is state-oriented if the dominant source of computation is the global state of the collective. More precisely, the computation is *state-determined* if it primarily depends on the known state of the system, with the communication-flow being at most the enabler of the information spreading across the collective (i.e., communication only refines local field updates). In this regime, the system behaviour is better interpreted as the evolution of a distributed spatiotemporal function over states rather than as message exchange between processes. An example of this perspective is that of field-based (aggregate) computing models, where the global field is the primary semantic object and communication is secondary or implicit.

Both communication and state oriented paradigms can thus be understood as orthogonal restrictions of the same underlying event-based semantics. The difference lies not in the event structure, but in which perspective is used as main abstraction to model the group behaviour (or some part of it).

VI. Conclusion

In this work, we characterise macroscopic programming in the attempt to unify different programming paradigms under the same hood. Specifically, we identify the fundamental abstractions that enable the creation of programs addressing the collective behaviour of a group. Our investigation supports the claim that various approaches can be, indeed, described in the general framework of macroscopic programming, thus simplifying their comparison and highlighting differences and similarities. We believe that this characterisation will help in describing new approaches, and that it will help newcomers in identifying the key principles of macroscopic programming.

References

[1] Audrito, G., Casadei, R., Torta, G.: A general framework and decentralised algorithms for collective computational processes. Future Gener. Comput. Syst. **158**, 11–27 (2024)

[2] Audrito, G., Damiani, F., Torta, G.: Composable models and guarantees for aggregate systems. Int. J. Softw. Tools Technol. Transf. **28**, 147–165 (2026)

[3] Brambilla, M., Ferrante, E., Birattari, M., Dorigo, M.: Swarm robotics: a review from the swarm engineering perspective. Swarm Intell. **7**(1), 1–41 (2013)

[4] Casadei, R.: Artificial collective intelligence engineering: A survey of concepts and perspectives. Artif. Life **29**(4), 433–467 (2023)

[5] Casadei, R., Viroli, M., Aguzzi, G., Pianini, D.: ScaFi: A Scala DSL and toolkit for aggregate programming. SoftwareX **20**, 101248 (2022)

[6] Casadei, R.: Macro-programming multi-agent systems: A framework for artificial collective intelligence. In: Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems. AAMAS, IFAAMAS. https://doi.org/10.65109/saxg2906

[7] Cruz-Filipe, L., Montesi, F.: A core model for choreographic programming. Theor. Comput. Sci. **802**, 38–66 (2020)

[8] D'Angelo, M., Gerasimou, S., Ghahremani, S., Grohmann, J., Nunes, I., Pournaras, E., Tomforde, S.: On learning in collective self-adaptive systems: State of practice and a 3d framework. In: 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). pp. 13–24. IEEE (2019)

[9] Farabegoli, N., Pianini, D., Casadei, R., Viroli, M.: Scalability through pulverisation: Declarative deployment reconfiguration at runtime. Future Gener. Comput. Syst. **161**, 545–558 (2024)

[10] Kuckling, J.: Recent trends in robot learning and evolution for swarm robotics. Frontiers Robotics AI **10** (2023)

[11] Montesi, F.: Introduction to Choreographies. Cambridge University Press, Cambridge (2023)

[12] Pinciroli, C., Beltrame, G.: Buzz: An extensible programming language for heterogeneous swarm robotics. In: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 3794–3800 (2016)

[13] Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From distributed coordination to field calculus and aggregate computing. J. Log. Algebraic Methods Program. **109**, 100486 (2019)

[14] Weisenburger, P., Wirth, J., Salvaneschi, G.: A survey of multitier programming. ACM Comput. Surv. **53**(4) (2020)