Macroprogramming: Concepts, State of the Art, and Opportunities of Macroscopic Behaviour Modelling

(Article begins on next page)

01 July 2024

Macroprogramming: Concepts, State of the Art, and Opportunities of Macroscopic Behaviour Modelling

ROBERTO CASADEI, Alma Mater Studiorum–Università di Bologna

Macroprogramming refers to the theory and practice of expressing the macro(scopic) behaviour of a collective system using a single program. Macroprogramming approaches are motivated by the need of effectively capturing *global/system-level* aspects and the *collective behaviour* of multiple computational components, while abstracting over low-level details. Previously, this programming style had been primarily adopted to describe the data-processing logic in *sensor networks*; recently, research forums on *spatial computing*, *collective systems*, and the *Internet of Things* have provided renewed interest in macro-approaches. However, related contributions are still fragmented and lack conceptual consistency. Therefore, to foster principled research, an integrated view of the field is provided, together with opportunities and challenges.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Software and its engineering** → **System description languages**; *Software system structures*; *Software development techniques*; • **Computing methodologies** → **Multi-agent systems**; **Distributed computing methodologies**.

Additional Key Words and Phrases: macro programming, system-level programming, collective intelligence

1 INTRODUCTION

Macroprogramming refers to the theory and practice of conveniently expressing the macro(scopic) behaviour of a system using a single program, often leveraging macro-level abstractions (e.g., collective state, group, or spatiotemporal abstractions). This is not to be confused with the use of macros (abbreviation for *macroinstructions*), i.e., the well-known mechanism for compile-time substitution of program pieces (e.g., characters, tokens, or abstract syntax trees), available in programming languages ranging from C and Common Lisp to Scala and Rust. Macros may be a mechanism for implementing macroprogramming, but not all uses of macros are macroprogramming, which concerns programming the overall behaviour of a system of multiple computational entities. Macro-programming is a paradigm driven by the need of designers and application developers to capture *system-level behaviour* while abstracting, in part, the behaviour and interaction of the individual components involved. It can be framed as a *paradigm* since it embodies a (*systemic*) view or perspective of programming, and accordingly provide *lenses* to the programmer for understanding and working on particular aspects of systems—especially those related to collective behaviour, interaction, and global, distributed properties.

Author's address: Roberto Casadei, roby.casadei@unibo.it, Alma Mater Studiorum–Università di Bologna, Via dell'Università 50, Cesena, Italy.

In the past, this style of programming has been primarily adopted to describe the behaviour of *wireless sensor networks (WSN)* [Mottola and Picco 2011], where data gathered from sensors are to be processed, aggregated, and possibly moved across different parts or regions of the network in order to be consolidated into useful, actionable information. More recently, certain research trends and niches have provided renewed interest in macro approaches. Research in the contexts of *Internet of Things (IoT)* and *cyber-physical systems (CPS)* has proposed macroprogramming approaches (cf. [Azzara et al. 2014; Mizzi et al. 2018]) to simplify the development of systems involving a multitude of connected sensors, actuators, and smart devices. In the *spatial computing* thread [Beal et al. 2012], space can represent both a means and a goal for macroprogramming. Indeed, declaring *what* has to be done *in* a spatiotemporal region allows systems to self-organise to effectively carry out the task at hand, dynamically adapting to the specifics of the current deployment and spatial positions of the components involved. Similarly, one can program a system, such as a drone fleet, in a high-level fashion to make it seek and maintain certain shapes and connectivity topologies. Indeed, swarm-level programming models have been proposed in robotics research [Pinciroli and Beltrame 2016]. In *distributed artificial intelligence (DAI)* and *multi-agent systems (MAS)* research [Adams 2001], an important distinction is made between the *micro* level of individual agents and the *macro* level of an "agent society", sometimes explicitly addressed by *organisation-oriented* programming approaches [Boissier et al. 2013]. In the field of *collective adaptive systems (CAS)* engineering [De Nicola et al. 2020; Ferscha 2015], macroprogramming abstractions can promote collective behaviour exhibiting self-* properties (e.g., self-organising, self-healing, self-configuring) [de Lemos et al. 2010; Kephart and Chess 2003]. In software-defined networking (SDN), the centralised view of the control plane has promoted programming networks as *"one big switch"* [Kang et al. 2013].

This work draws motivation from a profusion of macroprogramming approaches and languages that have been proposed in the last two decades, aiming to capture the aggregate behaviour of certain classes of distributed systems. However, contributions are sparse, isolated in research niches, and tend to be domain-specific as well as technological in nature. This survey aims to consolidate the state of the art, provide a map of the field, and foster research on macroprogramming.

This article is organised as follows. Section 2 covers the method adopted for carrying out the survey. Section 3 provides an overview of the research fields where macroprogramming techniques have been proposed, also tracing the history of the field. Section 4 defines a conceptual framework and taxonomy for macroprogramming. Section 5 is the core of the survey: it classifies and presents the selected primary studies. Section 6 provides an analysis of the surveyed approaches and discusses opportunities and challenges of macroprogramming. Section 7 covers related work, discussing the contributions of other secondary studies. Finally, Section 8 provides a wrap-up.

2 SURVEY METHOD

This section briefly describes how the survey has been carried out. It focusses on motivation, research questions, data sources, presentation of results, and terminology.

2.1 Survey Method

Though this is not a systematic literature review (SLR), the survey has been developed by considering guidelines by SLR methodologies like [Kitchenham and Charters 2007]. More details follow.

2.1.1 Review motivation. As anticipated in Section 1, the survey draws motivation by the emergence of a number of works that more or less explicitly identify themselves as macroprogramming approaches. Related secondary studies have been carried out in the past: they are reviewed in Section 7. However, they focus on particular perspectives or domains (e.g., spatial computing, or WSN programming), are a bit outdated, and consider macroprogramming as a particular class of

approaches in their custom scope. Critically, *macroprogramming has never been investigated as a field per se*, yet. Another major motivation lies in the *fragmentation* of macroprogramming-related works across disparate research fields and domains. Thus, a goal of this very survey is to provide a *map* of macroprogramming-related literature, promoting interaction between research communities and development of the field. More motivation is given by the urge of the following research questions.

2.1.2 Research goals and questions. The goal of this article is to explore the literature on macroprogramming *in breadth*, synthesise the major contributions, and provide a basis for further research. The focus is on the *programming* perspective, rather than e.g. modelling formalisms for analysis and prediction; namely, the contribution can be framed in *language-based software engineering* [Gupta 2015]. To better structure the investigation, we focus on the following research questions, inspired by the "six honest serving men" [Kipling 1902] as e.g., in [Flood 1994].

RQ0) *Why, where, and for who is macroprogramming most needed?*
RQ1) *What is macroprogramming and, especially, what is not?*
RQ2) *How is macroprogramming implemented? Namely, what are the main macroprogramming approaches and abstractions?*
RQ3) *What opportunities can arise from research on macroprogramming?*
RQ4) *What are the key challenges in macroprogramming systems?*

RQ0 is addressed in Section 3. RQ1 is addressed in Section 4. RQ2 is addressed in Section 5. Finally, RQ3 and RQ4 are addressed in Section 6.

2.1.3 Identification, selection, and quality assessment of primary research. Primary research studies have been identified by searching literature databases (such as Google Scholar, DBLP, IEEEXplore, ACM DL) for keywords such as "macroprogramming", "global-level programming", "network-wide programming", and "swarm programming", Terminology is fully covered and discussed in Section 2.2. Additional sources include other secondary and primary studies, which are surveyed in Section 7 and Section 5, respectively.

The survey scope is wide and includes PhD theses, technical reports, and papers presented at workshops, conferences, and journals as well as across different domains and research communities. Works that are deemed too preliminary (e.g., position papers), not enough "macro" (refer to Section 4), or neglecting the "programming" aspects (e.g., describing a middleware but no programming language) have been excluded, after being manually inspected.

2.1.4 Data extraction, synthesis, and dissemination. For each primary study, notes are taken regarding its *self-positioning* (i.e., how the authors define their contribution), its *programming model* (i.e., what main abstractions are provided), its *implementation* (i.e., how macro-abstractions are mapped to micro-level operations), and *source-code examples*. The data is syntheses using the conceptual framework introduced in Section 4. When covering and summarising the primary works in the survey (Section 5), we tend to keep and report the terminology originally used in the referenced papers, possibly explained and compared with the terminology used in this manuscript. This should help to preserve the richness and nuances of each work while the common perspective is ensured by proper selection and emphasis of the information included in the descriptions. Examples – adapted from those already included in the primary studies or created anew from composing code snippets described in those papers – are provided when they are reasonably "effective" or "diverse" from those already presented: i.e., they are brief and simple in transmitting how the reviewed approach looks like and works.

2.2 A Note on Terminology

A first issue in macroprogramming research is the fragmentation and ambiguity of terminology, which – together with domain fragmentation (see Section 3) – leads to (i) difficulty when searching for related work, and (ii) obstacles in the formation of a common understanding. Across literature, multiple terms such as macroprogramming, system-level programming, and global-level programming are used to refer to the same or similar concepts: this does not promote a unified view of the field and hinders progress by preventing the spread of related ideas. At the same time, there is a problem of usage of both over- and under-specific terms. Overly general terms both witness the lack and prevent the formation of a common ground. On the other hand, overly specific terms, mainly due to domain specificity of research endeavours, fail at recognising the general contributions or at advertising the effort in the context of a bigger picture.

In the following, we list some terms that have been used (or might be used) – with more or less good reason – when referring to macroprogramming, and analyse their semantic precision (by reasoning on their etymology and other common uses) as well as alternative meanings in literature (for conflicts with more or less widespread acceptations).

Macroprogramming, macro-programming, macro programming, macro-level programming. These are the premier terms for the subject of this article and may indeed refer to *programming macroscopic aspects of systems* (often, by leveraging macro-level abstractions). However, these terms are sometimes also used in other computer programming-related contexts. The potentially ambiguity stems from word "macro", which is and can be used to abbreviate both term "macroscopic" and term "macroinstructions"—often used in the sense of *macros*, i.e., the well-known programming language mechanism for compile-time substitutions of program pieces. Indeed, it is common to say that macros are written using a macro (programming) language. The result is that searching for these terms leads to a mix of results from both worlds. Unfortunately, being macros a very common mechanism [Lilis and Savidis 2020], macroscopic programming-related entries remain relatively little visible in search results, unless other keywords are used to narrow the context scope—but then, only a fragment of the corpus can be located.

System programming, system-level programming, system-oriented programming. All these terms are also ambiguous. Indeed, they strongly and traditionally refer to *low-level programming*, i.e., programming performed at a level close to the (computer) system (i.e., to the machine) [Appelbe and Hansen 1985]. System programming languages include, e.g., C, C++, Rust, and Go. A better name for these would probably be, as suggested by Dijkstra, *machine-oriented* languages, but such a "system" acceptation is a sediment of the field by now. The scarce accuracy of the term was also somewhat acknowledged by researchers in the object-oriented programming community [Nygaard 1997]. However, in some cases, system-level programming is contrasted with device-level programming, to mean approaches that address "a system as a whole" [Liang et al. 2016].

Centralised programming. This term [Gude et al. 2008; Lima et al. 2006] commonly refers to programming a distributed system through a single program where distribution is (partially [Waldo et al. 1996]) abstracted away, i.e., like if the distributed system were a centralised system, namely a software system on a single computer deployment. An example of centralised programming is *multi-tier programming* [Weisenburger et al. 2020]. This notion is certainly related to macroprogramming, since a "centralised perspective" where several distributed components can be addressed at once is a macroscopic perspective. However, as discussed in Section 4, programming the macro level often implies *more* than programming the individual components from a centralised perspective.

High-level programming. This term, identifying a style of programming that abstracts many details of the underlying platform, lacks of precision. Macroprogramming is a form of high-level programming, but not all the high-level programming is macroprogramming (for a conceptual framework for macroprogramming, refer to Section 4).

Domain-specific or alternative terminology: global-level programming, network-wide programming, organisational programming, swarm programming, aggregate programming, ensemble programming, global-to-local programming, team-level programming, organisation-oriented programming etc. These terms will be explained and properly organised in the following sections. From this list of terms, however, it is already possible to get a sense of (i) an intimate need, from different research communities, to linguistically emphasise a focus on macroscopic aspects of systems, and (ii) the urge for a common conceptual framework where such disparate contributions can be framed.

3 SCOPE AND HISTORICAL DEVELOPMENT OF MACROPROGRAMMING

Here, we provide an overview of the main research fields and application domains where macroprogramming techniques have been proposed, also tracing a historical development of the paradigm.

To the best of our knowledge, the term "macroprogramming" – in the acceptation used in this manuscript – first appeared in [Newton and Welsh 2004]. Multiple occurrences of the term appeared in other papers published later in the same year and in the following ones.

3.1 Wireless Sensor and Actuator Networks (WSAN)

WSANs are networks of embedded units capable of processing, communication, and sensing and/or acting [Mottola and Picco 2011]. They are a technology providing relatively low-cost monitoring and control of physical environments. Given the large number of involved devices, and the reasonable levels of heterogeneity and dynamicity for a given application, it became apparent that a benefit could be provided by high-level programming models abstracting from a series of low-level network details while still seeking to preserve efficiency. When a system consists of a large number of rather homogeneous entities, individuals tend to become less important to the functionality (while may well contribute to non-functional aspects): a WSN with 50 devices might perform worse than a 100-devices network, but these two networks can be programmed the same. Additionally, developers and researchers started realising that the individual sensors are actually a *proxy* or a *probe* for more important application abstractions such as information, streams, and events. At a next step, those abstractions started to become more high-level, and to address larger portions of the system beyond individual sensors, such as neighbourhoods [Whitehouse et al. 2004], or regions [Welsh and Mainland 2004]; accordingly, abstractions related to those more coarse-grained entities emerged, denoting contexts, aggregate views, fields—increasingly non-local abstractions. Among the high-level approaches, languages providing a *centralised view* of the WSN emerged; then, the step to macroprogramming was short. This is, indeed, one of the first domains where macroprogramming was introduced. Early works like TinyDB [Madden et al. 2002], Pieces [Liu et al. 2003], Abstract Regions [Welsh and Mainland 2004], and Regiment [Newton et al. 2007] are among the first contributions explicitly defining themselves as macroprogramming. A survey on macroprogramming for WSNs can be found in [Mottola and Picco 2011].

3.2 Spatial Computing

Space is generally important in ICT systems. This has been especially motivated and investigated in the *Computing Media and Languages for Space-Oriented Computation* seminar in Dagstuhl [dag 2007], where three key issues are found to be recurrent in many computer-based applications: *(i) coping with space*, for efficiency in computation; *(ii) embedding in space*, as in embedded and pervasive

computing; and *(iii) representing space*, for spatial awareness. What became apparent, also from WSN programming research, is that devices situated in space can become *representatives* of the spatial region they occupy and of the corresponding context. In this view, distributed systems and networks can be seen as *discrete approximations of continuous space-time regions and behaviours* [Bachrach et al. 2010]. Therefore, macroprogramming abstractions may abstract individual devices and rather focus on spatial patterns that such devices should cooperatively (re-)create—e.g., for morphogenesis [Jin and Meng 2011]. In general, dealing with *situated systems* [Lindblom and Ziemke 2003] – i.e., systems where components have a location in and coupling with (logical or physical) space, with typical corresponding consequences such as partial observability and local (inter)action – is simplified when recurring to spatial abstractions such as, e.g., computational fields [Mamei et al. 2004]. Spatial computing approaches are extensively surveyed in [Beal et al. 2012] (see Section 7 for details on the study) and include exemplars of macroprogramming such as Regiment [Newton et al. 2007] and MacroLab [Hnat et al. 2008].

3.3 Internet of Things, Cyber-Physical Systems, Edge-Fog-Cloud Computing Systems

The *Internet of Things (IoT)* [Atzori et al. 2010] refers to a paradigm and set of technologies supporting interconnection of smart devices and the bridging of computational systems with physical systems—the latter element being emphasised also through term *Cyber-Physical Systems (CPS)* [Serpanos 2018]. IoT systems share many commonalities with WSANs, so it is not surprising that contributions from the latter field have been extended to address IoT application development. Actually, the IoT can be considered as a superset of WSANs, with additional complexity due to the exacerbation of issues like heterogeneity, mobility, topology, dynamicity, infrastructural complexity, as well as functional and non-functional requirements. However, an IoT system can still be considered as a collective of interconnected smart devices, amenable to be considered by a macroscopic perspective.

Moreover, IoT systems tend to be more heterogeneous and infrastructurally rich, comprising edge, fog, and cloud computing layers [Yousefpour et al. 2019] to support various requirements including low-latency and low-bandwidth consumption. Interestingly, also the edge, the fog, and the cloud can be considered as computational (eco-)systems programmable at the macro-level [Pianini et al. 2021a]. This idea also underlies orchestration approaches based on Infrastructure-as-Code [Morris 2016], which can be considered a form of centralised, declarative programming.

Examples of IoT/CPS macroprogramming approaches include PyoT [Azzara et al. 2014], DDFlow [Noor et al. 2019], and MacroLab [Hnat et al. 2008], whereas preliminary approaches also considering edge/fog/cloud comprise ThingNet [Qiao et al. 2018].

3.4 Swarm robotics

A set of interacting robots can work as a collective, also known as a *swarm*. In this case, the focus of external observers tends to shift from the activity of individual robots to the activity of the swarm as a whole. Various tasks make sense at such a macro-perspective. For instance, we could ask a swarm to: move in flock formation towards a destination; split and later merge for avoiding a large obstacle; use, in a coordinated way, the sensing capabilities to estimate physical quantities (e.g., the mean temperature in a certain area) or other indicators (e.g., the risk of fire in a forest); or use, in a coordinated way, sensing and actuation capabilities to efficiently perform actions and tasks (e.g., quickly collecting toxic waste in industrial plants) possibly going beyond individual capabilities (e.g., moving heavy objects). Another prominent sub-field in robotics with emphasis on macroscopic features is modular, morphogenetic robotics [Jin and Meng 2011; Zykov et al. 2007], which considers collections of building-block modules that should dynamically self-reconfigure into functional shapes in order to address tasks, change, or damage. Indeed, the overall morphology of a modular swarm is a macro-level structure that must be dynamically sought through activity and

cooperation of the individual robots. The traditional question is: how can the individual robots be programmed such that the desired overall shape is produced? By a macroprogramming perspective, this question turns into: how can a swarm *as a whole* be programmed such that the overall shape is produced? Of course, this ultimately entails a definition of the behaviour of the individuals as well; however, the idea is to encapsulate the complexity of such a collective behaviour at the middleware level, behind proper macroscopic abstractions. Examples of macroprogramming languages for swarm robotics include Meld [Ashley-Rollman et al. 2007] (for modular robotics), Voltron [Mottola et al. 2014] (for drone teams), Buzz [Pinciroli and Beltrame 2016], TeCoLa [Koutsoubelias and Lalis 2016], and WOSP [Varughese et al. 2020] (for elementary robots).

3.5 Complex and Collective Adaptive Systems

Complex and collective adaptive systems (CAS) are collectives (i.e., collections of individuals) exhibiting a non-chaotic behaviour that is adaptive to the environment and cannot be (easily) reduced to the behaviour of the individuals, but that rather *emerges* from complex networks of situated interactions. These kinds of systems were originally observed in nature, but researchers have tried to bring those principles and ideas for development artificial, ICT-based CASs [De Nicola et al. 2020; Ferscha 2015]. The field of CAS engineering emerges from swarm computational intelligence [Kennedy 2006] and autonomic, self-adaptive computing [de Lemos et al. 2010; Kephart and Chess 2003]. The goal of CAS programming is to program the collective adaptive behaviour of a system. In general, two approaches are possible: *local-to-global*, where local behaviour is specified in order to promote emergence of a target global behaviour; or *global-to-local*, where the idea is to specify the intended global behaviour and come up with a mechanism to synthesise the corresponding local behaviour.

Since the notion of a *collective* (also known as *ensemble*) is per se a macro-level abstraction, it is natural to adopt macroprogramming techniques. Examples are provided in Section 5 and include ensemble-based approaches such as DEECo [Bures et al. 2013] and SCEL [De Nicola et al. 2014], and aggregate programming [Beal et al. 2015].

3.6 Other domains

In the following domains, macroprogramming has not actually been proposed explicitly, but similar needs can be perceived and very related ideas have indeed been considered.

3.6.1 Software-defined networking. Software-defined networking (SDN) [Kreutz et al. 2015] is an approach for the management of computer networks based on the idea of separating the data plane (forwarding) and the control plane (routing). Thanks to this separation, network devices become just entities responsible for forwarding, whereas control logic can be logically centralised in a single component. This logical centralisation directly leads to centralised programming (cf. Section 2.2) and hence to a macroprogramming viewpoint. This is also visible in the editorial note [Beckett et al. 2019], which provides a brief historical reflection on the development of such a vision, also known as *network-centric* or *network-wide programming* [Martins and McCann 2017].

Examples of network-wide programming include NetKAT [Anderson et al. 2014] and SNAP [Arashloo et al. 2016].

3.6.2 Parallel Programming and High-Performance Computing (HPC). Literature on parallel programming does include some germs of macroprogramming ideas, even though the focus on performance and low-level system programming arguably has hindered adoption of high-level abstractions. However, these can be found in parallel, *global-view* languages, such as those implementing the Partitioned Global Address Space (PGAS) model [Wael et al. 2015], where, e.g., directives have been proposed to represent *"high-level expressions of data distributions, parallel data movement,*

Ref.	Definition
[Bakshi and Prasanna 2005]	*"The objective of macroprogramming is to allow the programmer to write a distributed sensing application without explicitly managing control, coordination, and state maintenance at the individual node level. Macroprogramming languages provide abstractions that can specify aggregate behaviors that are automatically synthesized into software for each node in the target deployment. The structure of the underlying runtime system will depend on the particular programming model."*
[Whitehouse et al. 2006]	*"Macroprogramming is a term often used to refer to the process of writing a program that specifies global network behavior as opposed to the behavior of individual nodes."*
[Awan et al. 2007]	*" Macroprogramming specifies aggregate system behavior, as opposed to device-specific programs that code distributed behavior using explicit messaging. [...] Composing applications with reusable components allows the macroprogrammer to focus on application specification rather than low-level details or inter-node messaging."*
[Sookoor 2009]	*"Macroprogramming provides the user with the illusion of programming a single machine by abstracting away the low-level details of message passing and distributed computation."*
[Pathak and Prasanna 2011]	*"In macroprogramming, abstractions are provided to specify the high-level collaborative behavior at the system level, while intentionally hiding most of the low-level details concerning state maintenance or message passing from the programmer"*

Table 1. Some descriptions of macroprogramming from the literature.

processor arrangements and processor groups". Indeed, addressing the behaviour of multiple processors in terms of macroscopic patterns rather than in terms of micro-instructions could simplify programmability and still reach good performance through smart global-to-local mapping.

Other elements of similarities can be traced between Valiant's *Bulk Synchronous Parallel (BSP)* model [Valiant 1990] and the execution model of macroprogramming approaches such as aggregate computing [Beal et al. 2015], where multiple parallel processors work in *supersteps* involving communication and computation as specified by a single global program. Moreover, this tendency towards programming by a macroscopic perspective has been witnessed by some BSP-based models. For instance, in the domain of graph-processing, as discussed in the paper *From "Think Like a Vertex" to "Think Like a Graph"* [Tian et al. 2013], the Giraph++ framework has been proposed by replacing the vertex-centric model of Giraph with a *graph-centric model* to provide efficiency benefits by directly exposing graph partitions and optimising communications.

4 A CONCEPTUAL FRAMEWORK AND TAXONOMY

In this section, after some preliminaries (Section 4.1), we define macroprogramming, describe its essential elements (Section 4.2), and distinguish it from other related notions like *centralised programming* (Section 4.3). Then, we propose a taxonomy and conceptual framework (Section 4.4) for classifying and studying the macroprogramming approaches surveyed in Section 5.

4.1 Preliminaries

Consider the problem of programming the behaviour of a computational system S composed of multiple computational entities. Let A and B be two different entities of that system. We have three main *modes* for *affecting* their behaviour in order to promote the behaviour or properties ascribable to the overall system S (which, as we will shortly see, is essentially the goal of macroprogramming).

(1) *Change their context (e.g., inputs).* The entities will be indirectly influenced by the different context. For instance, if A is a sensor, it might sense a different value, which may in turn affect B and so on.

(2) *Interaction (e.g., trigger/orchestrate their behaviour).* For instance, if A is an actuator, it might be commanded to act upon the environment, which may in turn affect B and so on.

(3) *Set their behaviour.* Part of the behaviour of A and B may be set or changed such that, when activated (e.g., in a reactive or proactive way), certain global outcomes will be produced.
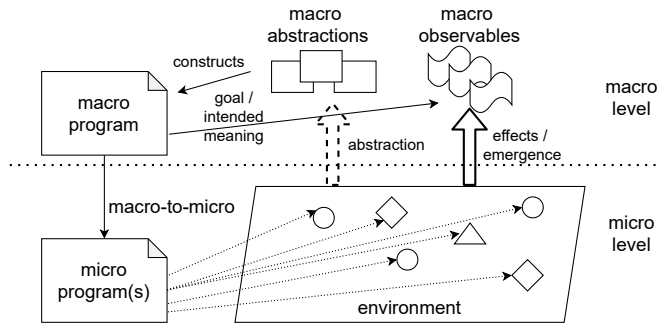
Fig. 1. The general idea of macroprogramming.

Let us use term *program* to mean an (abstract) description that can be *executed* by some (abstract) computational entity. Note that modes (1) and (2) allow a program to affect A or B, and hence $\mathcal{S}$, by having it executed by another entity, say C, assumed to be external to the arbitrary boundary of $\mathcal{S}$.

4.2 Macroprogramming: Definition and Basic Concepts

We define macroprogramming as *an abstract paradigm for programming the (macro)scopic behaviour of systems of computational entities*[1]. As a paradigm (see Section 4.3.2 for a discussion on this), it is "an approach to programming based on a mathematical theory **or a coherent set of principles**" [Van Roy 2009] (bold is added). Macroprogramming is based on the following principles, which can be partially extracted from the various definitions given in literature (cf. Table 1):

P1 *Micro-macro distinction.* Two main levels of a system are considered: a *macro* level (of global structures, of state, of behaviour) and a *micro* level (of computational entities).

P2 *Macroscopic perspective.* The programming activity tends to focus on macroscopic aspects of a *system*, which may include summary observations and views whereby micro-level entities are considered by a *global* (or *non-local*) and conceptually centralised perspective.

P3 *Macroprogram.* The output of the macroprogramming activity is a program that is conceptually executed by the system as a whole and whose intended meaning adopts the macroscopic perspective.

P4 *Macro-to-micro mapping.* A macroprogramming implementation has to define *how* a macroprogram is executed, by the system as a whole, which entails defining a *macro-to-micro mapping* logic—sometimes also known as *global-to-local* mapping [Hamann 2010]. I.e., from a macroprogram, micro-level programs or behaviours are derived or affected (cf. Section 4.1).

Figure 1 shows the general idea of the approach, graphically.

4.2.1 On micro-macro and local-global distinction. The micro-macro levels and the local-global scales usually used as equivalent concepts to distinguish smaller elements/scopes and larger elements/scopes somewhat "containing" or "being implied by" the former. The micro-macro distinction [Alexander 1987] (sometimes also space out by an intermediate, or *meso* level) is typical in many scientific areas including social sciences, systemics, and distributed artificial intelligence [Schillo et al. 2000] (cf. multi-agent systems [Wooldridge 2009]). For the sake of programming, just like a system (as an ontological and epistemological element) can be *defined* according to a boundary condition [Mobus and Kalton 2014], the distinction between two dimensions, micro and macro, is

---

[1]Possibly corresponding to physical devices through a notion of *digital twin* [Rasheed et al. 2020] or *physical computation* [Horsman et al. 2013].

similarly made through a design-oriented boundary or membership decision defining what belongs to one level or the other.

The intended meaning of macroprograms, and hence the ultimate goal of macroprogramming, seems to be related to the notion of *emergence* [Gignoux et al. 2017; Holland 1998; Kalantari et al. 2020; Wolf and Holvoet 2004]. In [Gignoux et al. 2017], the authors use graph theory to provide formal definitions of macroscopic states and microscopic states, and characterise emergence by analysing the general relationships between microscopic and macroscopic states.

What can we say, in general, about the entities at the micro and macro levels in macroprogramming? Micro entities have a computational behaviour, which may be autonomous (proactive), active, or reactive; and may or may not interact with other micro entities. So, for instance, data elements do not make for micro entities (they have no behaviour), while agents, actors, objects, and microservices do[2]. Regarding the macro level, we can distinguish between macro-level observables and macro-level constructs. A *macro-level observable* is a high-level observation of the system behaviour, i.e., a macro state as defined in [Gignoux et al. 2017], which is associated to the system as a whole and might be difficult to derive from micro state (the set of observations about the micro-level entities). The intended meaning, or goal, of a macroprogram, is generally a function of macro-level observables over some notion of time. A *macro-level construct* or *abstraction* is, instead, a description that can be mapped down to affect the behaviour of two or more micro-level entities (cf. Section 4.1). Implementing such a mapping is the macro-to-micro problem of macroprogramming.

4.2.2 On collectives. Macroprogramming usually targets so-called *collectives*—see Section 3. Term "collective" derives from Latin *colligere*, which means "to gather together". Typically [Masolo et al. 2020], a collective is an entity that gathers multiple *congeneric* elements together by some notion of *membership*. "Congeneric" means "belonging to the same genus", namely, of related nature. In other words, a collective is a group of similar individuals or entities that share something (e.g., a trait, a goal, a plan, a reason for unity, an environment, an interface) which justifies seeing them as a collective, overall. A group of co-located workers, a swarm of drones, the cells of an organ are examples of collectives, whereas a gathering of radically different or unrelated entities such as cells, rivers, and monkeys is not, intuitively. Being congeneric, the elements of a collective generally share goals and mechanisms for interaction and hence collaboration. The differences among the elements, often promoting larger collective capabilities by collaboration, may be due to genetic factors, individual historical developments, and the current environmental contexts driving diverse responses on similar inputs.

Heterogeneous collectives also exist (e.g., aggregates involving humans, autonomous robots, and sensors) and can be addressed by macroprogramming [Scekic et al. 2020]. However, heterogeneity tends to complicate macroprogramming by posing more importance on individuals' perspectives or widening the macro-to-micro gap—see Section 6.4.3 for a discussion.

4.2.3 On declarativity. A typical aspect of macroprogramming is *declarativity*. *Declarative programming* [Lloyd 1994] is a paradigm which focusses on expressing *what* the goal of computation is rather than *how* it must be achieved. Common and concrete aspects of a computation that can be abstracted away include the order of function evaluation (cf. functional programming), proving theorems from facts (cf. logic programming), and the specifics of data access (cf. query plans in databases and SQL). The general idea is to provide high-level abstractions capturing system-wide concerns by making assumptions promoting convenient mapping to component-level concerns. As such assumptions tend to be specific to an application domain, macroprogramming languages typically take the form of *domain-specific languages (DSLs)* [Beal et al. 2012].

---

[2]Possibly, even humans and other physical entities [Horsman et al. 2013].

4.3 What Macroprogramming Is (Not)

Programming essentially always deals with multiple interacting software elements, be them functions, objects, actors, or agents. Even though paradigms are more a matter of *mindset* and *abstractions*, rather than a matter of strict demarcation, a *demarcation issue* may be considered to better delineate a (nevertheless, fuzzy) boundary of macroprogramming. Macroprogramming is often centred around *macro-abstractions*: informally, constructs that involve, in some abstract way, (the context, state, or activity of) two or more micro-level entities. For instance:

- *macro-statements* (or *macro-instructions*), for imperative macroprogramming languages (e.g., "move the entire swarm to that target location", or "update the WSN state history to record the current temperature of the area");
- *macro-expressions*, evaluating to a macro-value (e.g., "the direction vector of the swarm towards the target location", "the mean temperature of the area covered by the network");

Other examples of macro-abstractions can be found in Section 6.2.

Consider the following artificial Scala program:

```scala
// Library code (non-macroprogramming)
object swarm {
  def robots = // ...
  def move(target: Pos): Unit = robots.foreach(robot => robot.move(target))
  def energyLevel(): Double = robots.map(_.energyLevel).sum / robots.size
  def positions(): Set[Pos] = robots.map(_.position)
  def monitor(area: Area): Unit = // ...
  // ...
}

// User code (macroprogramming)
if(swarm.energyLevel() < WARNING_ENERGY_LEVEL){
  swarm.move(rechargingStation())
} else {
  swarm.monitor(targetArea())
}
```

The swarm object provides a macro-abstraction over the set of underlying robots. Indeed, such a code might be written to abstract from a series of low-level details: the obstacle avoidance behaviour of individual robots; the fact that robots of the swarm move collectively in flock formation; the way sensors and actuators perceive distances to other robots, obstacles, and acceleration, to control stability and speed of each moving robot. The intended meaning of the program may refer to macro-observables that may or not may accessible by the program (cf. side-effects). The library code provides an implementation of the macroprogramming system. It maps the expressions of the user macro-program down to micro-level behaviour. Here, the macro-to-micro approach may be interpreted as an interaction mode – it is the running thread that interacts with the micro-level entities through the program control flow – or an execution mode – the macro-program is executed by the micro-level entities. This simplified example shows a macroprogramming language as an library/API within an existing host language, also called an internal DSL; actual examples of internal macroprogramming DSLs include Chronus [Wada et al. 2010] and ScaFi [Casadei et al. 2020b].

Doing macroprogramming is very much a matter of perspective. If the micro-macro distinction we are considering is robots vs. a swarm, then the library code (Lines 1-9), individually addressing each robot of the swarm with a specific instruction, is not macroprogramming, properly; vice versa, the user code (Lines 11-16), addressing the swarm as a whole, does represent an example of macroprogramming. However, the library code could be considered macroprogramming under a micro-macro viewpoint of sensors/actuators vs. a robot.

4.3.1 Weak vs. strong macroprogramming. In a nutshell, the central idea of macroprogramming is considering *the entire system as the abstract machine* for the operations. Notice that adopting a

centralised perspective to programming, where a centralised program has access to all the individual entities, is not generally sufficient for effective macroprogramming: there should typically be *at least one intermediate level of indirection*[3], where macro-operations turn into micro-operations. In the example above, while the library code can directly access the individual robots, the user code indirectly accesses them through the swarm macro-abstraction.

Essentially, *directly* feeding micro-operations to the micro-level entities or specifying the individual behaviours of the parts breaks the macroprogramming abstraction, or makes it *leaky* [Kiczales 1992; Spolsky 2004]. This is one reason (in addition to limited emphasis on behaviour) for which, e.g., formalisms for concurrent systems such as process-algebraic approaches [Baeten 2005], certain component-based approaches, and multi-tier programming [Weisenburger et al. 2020] are not generally considered macroprogramming. However, several approaches in literature defined themselves as macroprogramming despite basically embodying merely a form of centralised programming. Some of these may provide some macroprogramming abstractions (e.g., an object from which individual entities can be dynamically retrieved), but would nevertheless appear as a *weak* form of macroprogramming. We may consider the *macroscopic stance* as a degree, and hence define *strong* macroprogramming approaches those where *only* macro-abstractions are provided. For demarcation purposes, we propose to call those centralised programming approaches that inherently adopt a macro-level, global perspective but directly address individuals through micro-level instructions as *weak macroprogramming* or *meso-programming*. Considering the "macro-ness" as a continuum, and hence admitting that languages can be "more macro" or "less macro", allow us to be more comprehensive in these early stages.

4.3.2 Macroprogramming as a Paradigm. [Van Roy 2009] defines a *programming paradigm* as *"an approach to programming a computer[-based system] based on a mathematical theory **or a coherent set of principles**"* (bold is added). Van Roy classifies paradigms according to (i) whether or not they can express observable nondeterminism and (ii) how strongly they support state (e.g., according to whether it is named, deterministic, and concurrent). Also interesting is Van Roy's view of computer programming as a way to deal with complexity (e.g., number of interacting components) and randomness (non-determinism) to make aggregates (unorganised complexity) and machines (organised simplicity) into systems (organised complexity). Macroprogramming effectively deals with aggregates, turning them into programmable systems.

We argue the principles outlined in this section form sufficient ground for macroprogramming to be considered a paradigm, and hence aggregate multiple approaches under its umbrella. It is a paradigm in a way similar to *declarative programming* [Lloyd 1994], which is "concerned with writing down *what* should be computed and much less with *how* it should be computed" [Finkelstein et al. 2003]. Then, paradigms like functional and logic programming are considered as more specific forms of declarative programming. As shown in Section 5, also concrete macroprogramming languages can adopt a specific paradigm (e.g., functional, logic, or object-oriented).

The notion itself of a paradigm has sometimes been criticised in teaching programming [Krishnamurthi and Fisler 2019] for its fuzziness and coarse grain, preferring epistemological devices like notional machines [Fincher et al. 2020]. However, our stance is that the notion of a paradigm may still be useful as a lens or perspective for observing, comparing, and relating several concrete programming approaches, and as a core notion around which researchers on disparate topics can self-identify and connect through shared terms and ideas.

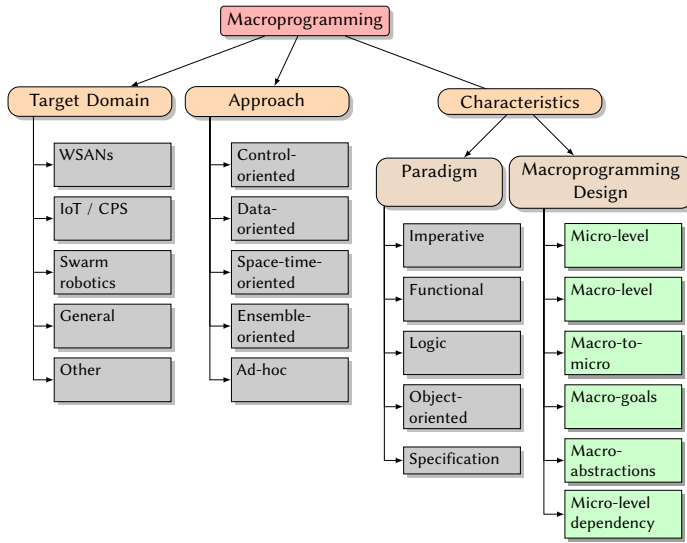[3]Informally, indirection refers to the ability to reference some object through another object.

Fig. 2. A taxonomy of macroprogramming.

4.4 Taxonomy

We propose to classify and analyse macroprogramming approaches according to the following elements, succinctly represented in Figure 2.

1) *Target domain.* It refers to the application domain explicitly addressed by a macroprogramming approach. This is relevant since domain-specific abstractions and assumptions are typically leveraged to properly deal with the abstraction gap induced by declarativity. Label "General" is used to indicate that an approach addresses distributed systems in general, whereas "Other" means that the approach addresses a specific domain different from the others.

2) *Approach.* We propose to classify macroprogramming languages according to the main approach they follow.
   - *Control-oriented.* Emphasis is on specifying control flow and instructions for the system.
   - *Data-oriented.* Emphasis is on specification of data and data flow.
   - *Space-time-oriented.* Emphasis is on specification of spatial, geometric, or topological aspects and their evolution over time.
   - *Ensemble-oriented.* Emphasis is on specification of organisational structures as well as tasks and interaction between groups of components.
   - *Ad-hoc.* The approach is peculiar and cannot be easily related with the previous ones.

3) *Characteristics.*

3a) *Paradigm.* The paradigm upon which macroprogramming abstractions are supported (the main one in case of multi-paradigm languages).

3b) *Macroprogramming design.* Elements characterising a particular macroprogramming language.
   - *Micro-level*: the individual components and aspects that collectively make up the system.
   - *Macro-level*: the system as a whole and its macroscopic aspects.
   - *Macro-to-micro*: the approach followed by macro-programs to affect micro-level behaviour. We distinguish four main modalities based on the discussion in Section 4.1: (i) *context*, where global state, inputs, or node parameters are set; (ii) *interaction*, where a process is used to orchestrate micro-level entities; (iii) *compilation*, where the macroprogram is

translated into the micro-programs; (iv) *execution*, where the macro-program is executed by the micro-level entities according to some (ad-hoc) execution model. .

- – *Macro-goals*: the objectives that macro-programs are meant to reach (typically, abstraction, flexibility, and optimisability—as a result of declarativity).
- – *Macro-abstractions*: the abstractions provided by a macroprogramming approach that are instrumental for achieving or capturing macroscopic aspects or goals of the system.
- – *Micro-level dependency*: the extent to which the macroprogramming language depends on micro-level components or aspects. We consider three levels: (i) **D**ependent (if micro-level elements are always visible), (ii) **I**ndependent (if micro-level elements are abstracted away), or (iii) **S**calable (if micro-level elements can be abstracted away as well as accessed, in case).

Elements of this taxonomy integrate and are partially inspired by some perspectives of previous work covered in Section 7.

Table 2. Summary of the surveyed macroprogramming approaches. The first column tells whether the approach is explicitly advertised as macroprogramming.

	Name/Ref.	Domain	Approach	Paradigm	Micro-level	Macro-level	Macro-to-micro	Macro-abstractions	Micro-dependency
●	Market-Based Macroprogramming [Mainland et al. 2004]	WSANs	ad-hoc	specification	nodes	WSN	context	virtual markets; good price	independent
●	BNM [Mamei 2011]	General	ad-hoc	specification	nodes	network	execution	Bayesian network tasks	dependent
	Graph-centric programming, Giraph++ [Tian et al. 2013]	Other	ad-hoc	imperative	nodes	graph	execution	subgraph	dependent
	NetKAT [Anderson et al. 2014]; SNAP [Arashloo et al. 2016]	Other	ad-hoc	imperative	switches	network	compilation	network state; network slices	dependent
	WOSP [Varughese et al. 2020]	Robotics	ad-hoc	specification	robots	swarm	execution	common behaviour	dependent
	PIECES [Liu et al. 2003]	Other	control-oriented	object-oriented	(sensor) nodes	WSAN	interaction	global state; state pieces; groups	dependent
●	Kairos [Gummadi et al. 2005]	WSANs	control-oriented	imperative	nodes	WSN	compilation	centralised view; node iterators; neighbourhoods; remote data access	dependent
●	PyoT [Azzara et al. 2014]	IoT/CPS	control-oriented	object-oriented	resources	IoT system	interaction	resource groups	dependent
	Buzz [Pinciroli and Beltrame 2016]	Robotics	control-oriented	imperative	robots	swarm	execution	swarm; neighbourhood; virtual stigmergy	dependent
	Dolphin [Lima et al. 2018]	Robotics	control-oriented	imperative	vehicles	vehicle network	interaction	vehicle sets; vehicle selection expressions	dependent
●	makeSense mPL [Mottola et al. 2019]	WSANs	control-oriented	object-oriented	nodes	WSN	compilation	distributed actions (report, tell, collective actions)	dependent
	Warble [Saputra et al. 2019]	IoT/CPS	control-oriented	object-oriented	things	IoT system	interaction	things selectors; bindings	dependent
	TinyDB [Madden et al. 2002]	WSANs	data-oriented	specification	nodes	WSN	compilation	database	independent
●	ATaG [Bakshi et al. 2013]	WSANs	data-oriented	specification	nodes	WSN	compilation	data flow graph	independent
●	Semantic Streams [Whitehouse et al. 2006]	WSANs	data-oriented	logic	nodes	WSN	execution	event streams; semantic services; inference units; regions	independent
●	Regiment [Newton et al. 2007]	WSANs	data-oriented	functional	nodes	WSN	compilation	time-varying signals; regions	scalable
●	COSMOS [Awan et al. 2007]	WSANs	data-oriented	specification	nodes	WSN	compilation	data flow graph	independent
●	Flask [Mainland et al. 2008]	WSANs	data-oriented	functional	nodes	WSN	execution	nfold macroprogramming combinator	independent
●	SOSNA [Karpinski and Cahill 2008]	WSANs	data-oriented	functional	nodes	WSAN	execution	streams of spatial values	scalable
●	MacroLab [Hnat et al. 2008]	IoT/CPS	data-oriented	imperative	nodes	CPS	compilation	macrovector; neighbourhoods	scalable
●	Nano-CF [Gupta et al. 2011]	WSANs	data-oriented	specification	nodes	WSN	execution	services; jobs	dependent
●	Pico-MP [Dulay et al. 2018]	WSANs	data-oriented	logic	nodes	WSAN	compilation	global formula on network data	dependent
●	D'Artagnan [Mizzi et al. 2018]; Porthos [Mizzi et al. 2019]	IoT/CPS	data-oriented	functional	IoT devices	IoT system	compilation	data streams	independent
●	DDFlow [Noor et al. 2019]	IoT/CPS	data-oriented	specification	IoT devices	IoT system	execution	data flow graph	independent

Table 2. Summary of the surveyed macroprogramming approaches. The first column tells whether the approach is explicitly advertised as macroprogramming.

	Name/Ref.	Domain	Approach	Paradigm	Micro-level	Macro-level	Macro-to-micro	Macro-abstractions	Micro-dependency
	MOISE [Hübner et al. 2007]	General	ensemble-oriented	specification	agents	multi-agent system	execution	organisations; roles; groups; missions	dependent
	Scopes [Jacobi et al. 2008]	WSANs	ensemble-oriented	specification	nodes	WSN	execution	scope (ensemble); scope membership	independent
●	EcoCast [Tu et al. 2011]	WSANs	ensemble-oriented	object-oriented	nodes	WSN	compilation	group handles; group-wide operations	dependent
	DEECo [Bures et al. 2013]	General	ensemble-oriented	specification	components	distributed system	execution	ensemble	dependent
	SCEL [De Nicola et al. 2014]; AbC [Alrahman et al. 2015]; CARMA [Loreti and Hillston 2016]; AErlang [De Nicola et al. 2018]	General	ensemble-oriented	specification	components	self-* system	execution	ensembles; group-oriented communication	dependent
●	Voltron [Mottola et al. 2014]	Robotics	ensemble-oriented	imperative	drones	swarm	compilation	teams; spatially situated tasks	dependent
	Comingle [Lam et al. 2015]	General	ensemble-oriented	logic	app nodes	distributed system	compilation	collective information; system state evolution	dependent
	TECOLA [Koutsoubelias and Lalis 2016]	Robotics	ensemble-oriented	object-oriented	robots	robotic team	execution	team-level services; mission groups; membership rules	dependent
	PaROS [Dedousis and Kalogeraki 2018]	Robotics	ensemble-oriented	object-oriented	robots	swarm	execution	abstract swarms; path planning; task partitioning	dependent
	Aggregate Programming [Viroli et al. 2019]; Proto [Beal and Bachrach 2006]; Protelis [Pianini et al. 2015]; ScaFi [Casadei et al. 2020b]	General	ensemble-oriented	functional	devices	distributed system	execution	computational fields; neighbourhoods; macro-behaviour functions	scalable
	SmartSociety [Scekic et al. 2020]	General	ensemble-oriented	object-oriented	human and machine peers	socio-technical system	interaction	collectives; collective-based tasks	independent
	Abstract Regions [Welsh and Mainland 2004]	WSANs	space-time-oriented	imperative	nodes	WSN	compilation	regions; region-aware data access	independent
	SpatialViews [Ni et al. 2005]	General	space-time-oriented	imperative	devices	MANET	interaction	spatial views (virtual networks)	dependent
●	STOP [Wada et al. 2007]; Chronus [Wada et al. 2010]	WSANs	space-time-oriented	object-oriented	nodes	WSN	interaction	space-time slices	independent
	Meld [Ashley-Rollman et al. 2007]	Robotics	space-time-oriented	logic	modular robots	robot ensemble	compilation	collective information; collective deduction	independent
●	Sense2P [Choochaisri et al. 2012]	WSANs	space-time-oriented	logic	nodes	WSN	execution	logical rule	independent
	PLEIADES [Bouget et al. 2018]	General	space-time-oriented	functional	nodes	distributed system	execution	shape templates	dependent

5 MACROPROGRAMMING APPROACHES

This section provides a survey of macroprogramming languages, which are analysed as per the conceptual framework of Section 4. The contributions are classified and organised as per the approach classes proposed in Section 4.4. A summary of the survey is provided in Table 2.

5.1 Control-oriented approaches

Control-oriented approaches emphasise an *imperative* macroprogramming style where control flow is specified and/or explicitly controlled for the system and instructions are issued to query or act on system components. This contrasts with data-driven approaches where control flow is a consequence of relationships among data. With control orientation, implicit or explicit sequences, conditionals, and loops may be used to describe what the macro-system or its components have to perform.

Representative example: Kairos [Gummadi et al. 2005]. It is a procedural macroprogramming language for WSNs that assumes loose synchrony and leverages eventual consistency to keep low overhead. The approach is *control-driven* and *node-dependent*—i.e., nodes and node state are explicitly manipulated at the programming level. In Kairos, the programmer writes a centralised program expressing the global specification of a distributed computation, which is compiled to a node-specific program. Kairos exposes three main abstractions: addressing of arbitrary nodes (e.g., by names or iterators like `node_list`), inspection of one-hop neighbour nodes (e.g., via function `get_neighbors`), and remote data access at nodes (e.g., with expressions `variable@node`). As an example, consider a simple self-healing hop-gradient computation, i.e., an algorithm that makes each node in the system yield the corresponding hop-by-hop distance towards a root node [Audrito et al. 2017].

```
1  node_list nodes = get_available_nodes();
2  int dist;
3  for(node n = get_first(nodes); n!=NULL; n=get_next(nodes)){
4    // Initialisation
5    if(n==root){ dist = 0 } else { dist = INF };
6    // Event loop
7    for(;;){
8      sleep(sleep_interval);
9      node_list nbrs = get_neighbors(n);
10     for(node nbr = get_first(nbrs); nbr!=NULL; nbr=get_next(nbrs){
11       if(dist@nbr+1 < dist){ dist = dist@nbr+1; }
12 } } }
```

Concerning macro-to-micro mechanics and implementation, during the translation of the macro-program into node-level programs, references to remote data are expanded into calls to the *Kairos runtime*, a software component which is assumed to be available in every node of the system. Specifically, the Kairos runtime deals with *managed objects* (objects owned by a node that are to be made available to remote notes) and *cached objects* (local views of managed objects owned by remote nodes), through asynchronous hop-by-hop communication—contrast this with synchronous data access calls in Kairos programs. Issues at the middleware level include supporting end-to-end reliable routing and management of dynamic topologies.

5.2 Data-oriented and database abstraction approaches

Data-oriented approaches define the macro-level behaviour of a system in terms of goals and activities of data gathering and processing. Sometimes, this is taken to the extreme, considering the system as a kind of distributed database keeping spatiotemporal or aggregated data.

Representative example #1: TinyDB [Madden et al. 2002]. TinyDB is a query processing system that considers a WSAN as a database. TinyDB supports an SQL-like language for expressing queries and actuations. A query looks like the following:

```
1  SELECT nodeId, temperature  WHERE temperature > k FROM sensors
2  SAMPLE PERIOD 5 minutes
```

Therefore, the approach is fully declarative and the system must find itself a strategy to map the global goal to local behaviour of the sensor nodes. We remark that the *behaviour* of the individual nodes is driven partly by the query-like macroprogram and partly by a basic "execution protocol" (providing a structure for the emergence of global behaviour) which is the same for all the nodes. Nodes work in *epochs*, corresponding to sampling periods, in a synchronised fashion. They sleep for most of the time; they wake up to sample sensors, gather neighbour data, process data, and send results to their parent node. This execution protocol is very similar to those used by other macroprogramming approaches, such as aggregate computing [Viroli et al. 2019] which is a paradigm for self-organising systems of agents.

Representative example #2: Semantic Streams [Whitehouse et al. 2006]. Semantic Streams is a logic-based, declarative language for expressing semantic queries over WSN data. It builds on two main abstractions: *event streams* and *inference units* (processes on event streams). For instance, the following program

```
1  stream (Y), isa (Y, histogram),   % histogram events
2  property (Y , X, stream),         % a histogram event has a stream X
3  property (Y , time, property),    %   and a time property
4  stream (X), isa(X, objectDetected),  %  stream X consists of object detection events
5  property (X, [[0,0,0],[50,50,0]], region). % ... within a given region
```

can be used to query for and plot `objectDetected` events in a given area across time. The macroprogramming system implementation is based on service composition and embedding. The query planner builds a task graph to be deployed to individual nodes, which will dynamically instantiate services, resolve conflicts between tasks and resources, and execute the queries.

5.3 Space-time oriented approaches

Space-time-oriented macroprogramming approaches are those that leverage spatial and temporal abstractions to organise the behaviour of a system. These approaches work by defining ways to connect devices (or their data, activities, and interactions) to space-time locations or regions.

Reference example #1: SpatialViews [Ni et al. 2005]. This approach works by abstracting a MANET into *spatial views* (i.e., collections of *virtual nodes*) of a configurable space-time granularity, that can be iterated on to visit nodes and request services. In detail, the model is as follows. A physical network consists of physical nodes. A physical node has a spatio-temporal location and a set of provided services. A virtual node is the digital twin of a physical node: its programming abstraction. A spatial view defines a virtual network over the physical network which is discovered and instantiated when iterated. Operationally, the system works by migratory execution of the program during iteration. The SpatialViews language is implemented as an extension to Java.

```
1  // Spatial views are collections of virtual nodes
2  spatialview sv1 = Camera @ BuildingC.Floor3;
3  spatialview sv2 = TemperatureSensor @ CampusB % 50; // 50 meters space granularity
4
5  // Discover virtual nodes in a spatial view
6  visiteach x : sv1 every 5 forever { x.getPicture().upload(); }
7
8  // Take average of temperatures
9  sumreduction float s = 0;
10 sumreduction int n = 0;
```

```
11  visiteach y : sv2 { s += y.read(); n++; }
12  float avg = s/n;
```

Space-time granularities are used to distinguish virtual nodes, which are visited once per iteration; instead, the underlying physical nodes might be visited more than once (e.g., because of mobility or after a quantum of time granularity). We remark that this work did not use any "macroprogramming"-like term to label SpatialViews, though clearly embracing the paradigm.

Reference example #2: SpaceTime Oriented Programming (STOP) [Wada et al. 2007], a.k.a. Chronus [Wada et al. 2010]. This WSN macroprogramming system exposes a spacetime abstraction to support collection and processing of past or future data in arbitrary spatio-temporal resolutions. Architecturally, it consists of a network of battery-powered sensors (where data is gathered) and base stations (where data is processed) linked to a gateway connected to the STOP server, which holds network data in the so-called *spatiotemporal database*. Operationally, the system is implemented through mobile agents carrying data to the STOP server, which in turn updates the database: *event agents* detect events and replicate themselves to move hop-by-hop towards a base station, where they finally *push* data; by contrast, *query agents* move across a spatial region in order to *pull* relevant data. The STOP/Chronus language is an object-oriented, Ruby DSL enabling on-command and on-demand (event-driven) data collection and processing. An example, selected and adapted from [Wada et al. 2007], is the following.

```
1  sp = Spacetime.new(Polygon.new(points), RelativePeriod.new(NOW, Hr-1))
2  spaces = sp.get_spaces_every(Min 5, Sec 10, 80)
3  values = spaces.collect { |space|
4    space.get_data('f-spectrum', MAX, Min 2){
5      |event_type, value, space, time |
6      # ...
7  } }
```

This program queries data in space-time "slices" that abstract the data generation activity of the underlying collection of sensor nodes. Indeed, it focusses on a macroscopic perspective.

5.4   Collective adaptive systems and ensemble-based approaches

Macroprogramming is also popular in the field of multi-agent (MAS) [Wooldridge 2009] and collective adaptive systems (CAS) [Ferscha 2015] engineering. CASs approaches are quite related to spatiotemporal approaches since CASs are often situated and space represents a foundational structure for coordination. In these approaches, it is common to consider large, dynamic groups of devices as first-class abstractions, which are commonly referred to as *ensembles*, *collectives*, or *aggregates*. The general idea is to support interaction between (sub-)groups of devices by abstracting certain details away (e.g., membership, connections, concurrency, failure). With respect to the network abstraction and other macroprogramming approaches, the works focus more on addressing the specification of dynamic ensembles, do not take an explicit, spatial space or are not limited to data gathering and processing.

Reference example on CAS programming: Aggregate programming [Viroli et al. 2019]. Aggregate programming is a macroprogramming paradigm, founded on *field calculi* [Viroli et al. 2019], for programming CASs. It builds on the *computational field* abstraction, a conceptually distributed data structure that maps any device of a system to a value, over time. Then, macroscopic behaviour can be expressed in terms of a single program which manipulates fields through constructs for state management, neighbourhood-based interaction, and domain partitioning (i.e., the ability to run a computation on a subset of the system nodes). Aggregate programming is supported by languages such as the Scala-internal DSL ScaFi [Casadei et al. 2020b] and the standalone DSL Protelis [Pianini

et al. 2015]. For instance, the problem of counting, in any device, the number of neighbour devices experiencing a high temperature can be expressed in ScaFi as follows:

```
1  foldhood(0)(_+_)(if(nbr(sense("temperature"))) 1 else 0)
```

where `foldhood(init)(acc)(f)` folds over the neighbourhood of each device by aggregating the neighbours' evaluation of `f` through accumulation function `acc`, starting with `init`. The interesting aspect about aggregate programming is that it is possible to capture collective behaviour into reusable *functions* (from which libraries of domain-specific features can be defined) and *compose* functions "from fields to fields" to define increasingly complex behaviour. For instance, the following `channel` functionality reuses functions provided by the ScaFi library to build a minimum-width path field from a source to a destination device, which is – crucially – able to self-adapt to input changes (i.e., different source or destination) and topology changes (e.g., as devices move or leave the system).

```
1  // source: input Boolean field (true only in the source device)
2  // target: input Boolean field (true only in the target device)
3  // width: input floating-point field for enlarging the channel
4  def channel(source: Boolean, target: Boolean, width: Double): Boolean = {
5    distanceTo(source)+distanceTo(target) <= distanceBetween(source,target)+width
6  } // output: true if the device belongs to the channel, false otherwise
```

Notice how this program abstracts from the individual devices at the micro-level: such a `channel` function denotes a macro-level structure that is sustained by repeated computation and interaction from the underlying network of devices. In virtue of this flexibility, aggregate programming can be deemed a *scalable* macroprogramming approach as it retains the ability to address individual devices but provides tools for raising the abstraction level.

Reference example for ensemble-based programming: PaROS (PROgramming Swarm) [Dedousis and Kalogeraki 2018]. PaROS is a framework for programming swarms of robots. It proposes an *abstract swarm* abstraction, implemented through a Java API, to promote swarm orchestration and spatial organisation. The API consists of functions for: path planning, declaration of points of interest or spatial areas to be inspected, enumeration of the robots in the swarm, task partitioning, setting handlers for detection events or robot failure. A program in PaROS looks like the following.

```
1  // Build a swarm from a set of drones
2  Swarm swarm = new Swarm(setOfDrones);
3  // Create flight plans by splitting an area and assigning sub-areas
4  swarm.areaDeclaration(targetArea);
5  // Define a collective task
6  swarm.setTask(Task.COVERAGE);
7  // Adds a handler for event detection
8  swarm.eventHandler((drone) -> { System.out.print("Event detected by " + drone); });
9  // Starts the mission: will run pathPlanning() and droneManipulation()
10 swarm.startMission();
11 // While the mission is running...
12 while(swarm.isMissionRunning()){
13   for(Drone drone : swarm.getListOfDrones()){
14     if(drone.isTaskComplete()){
15       doSomethingWith(drone.getCameraImage(camera));
16 } } }
```

Many details regarding the coordination of the swarm are abstracted away. Therefore, PaROS promotes a multi-paradigm approach comprising elements from imperative, declarative, and event-driven programming.

5.5 Ad-hoc approaches

Ad-hoc approaches are those that make very peculiar assumptions on the programming model or on the underlying system.

For instance, in Market-Based Macroprogramming (MBM) [Mainland et al. 2004], a sensor network is programmed as a *virtual market*. The nodes of the network follow a fixed behaviour protocol where they "sell" *actions* to get a *profit*. They choose actions according to a local *utility function* that expresses a trade-off between the profit and the *cost* of performing the action.

Another example is Wave-Oriented Swarm Programming (WOSP) [Varughese et al. 2020], an approach for swarm-level programming that requires minimalistic communication, inspired by two biological mechanisms: (i) scroll waves in slime mould and (ii) periodic light emission in fireflies. Each robot of the swarm follows a protocol where it is initially *inactive*, listening for incoming pings; upon reception of a ping, it runs a "relay code block" and goes into an *active* state where it emits a ping; after the emission of a ping, it goes in the *refractory state*, where it does nothing, being insensible to pings, and finally turns back to the inactive state after a refractory period.

Other examples are given by languages for Software-Defined Networking (SDN), like NetKAT [Anderson et al. 2014], SNAP (Stateful Network Abstractions for Packet processing) [Arashloo et al. 2016]. These consider the network as *"one big switch"* [Kang et al. 2013] with state. The NetKAT language is based on KAT (Kleene Algebra with Tests) plus constructs for networking. Conceptually, a macro-program in these languages is a function of a packet and network state (represented through global variables) that produces a set of packets and a new network state as output. In practice, a program consists of the classical imperative constructs (assignment, conditionals, loops) which are however interpreted in the SDN domain. The compiler translates the macro-program into micro-programs for the network devices dealing with *traffic routing* and *placement of state variables*.

6 ANALYSIS AND OUTLOOK

In this section, we analyse some data from the survey (Section 6.1), the surveyed approaches by a technical point of view (Section 6.2), and then review significant opportunities (Section 6.3) and challenges (Section 6.4) related to macroprogramming.

6.1 Data and Trends

In this survey, we have considered a total of 66 *works*, and have *included* (i.e., considered as a macroprogramming approach, after manual analysis) 49 *works*, of which 39 *core* works have been identified (i.e., some approaches have been implemented through multiple published languages or DSLs) corresponding to the number of rows of Table 2.

The distribution of the included works by (publication) year is reported in Figure 3a. From this histogram, we observe the rise of macroprogramming from WSN research in early 2000s, a loss of hype in early 2010s, and a new wave from 2014 as a result of recent trends and developments in fields like the IoT, CPSs, and CASs (cf. Section 3). The distribution of works across domains is shown in Figure 3b, where we observe a predominance of the WSN domain; the domain fragmentation seems to be a characteristic of the second wave of macroprogramming. Another interesting datum is how many of the surveyed works explicitly advertise themselves as macroprogramming: according to Figure 3c, this is only the case for 18 out of 39 core works.

Another significant aspect concerns the availability of accessible software for a macroprogramming language. According to Figure 3d, the number of works for which a repository or website exists that provides access to software is 18 out of 49 works. Arguably, this low score is partially due to the limited practice of providing artifacts in early 2000s, as well as to the obsolescence of some of the proposed languages from those years.
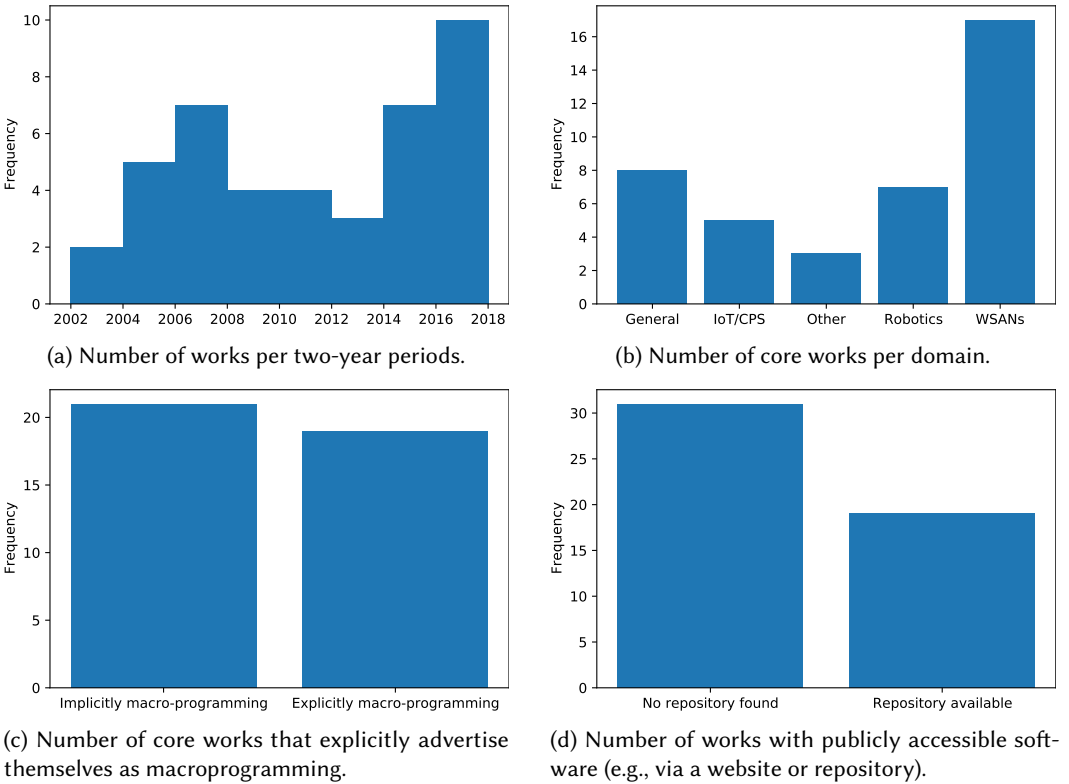
(a) Number of works per two-year periods.

(b) Number of core works per domain.

(c) Number of core works that explicitly advertise themselves as macroprogramming.

(d) Number of works with publicly accessible software (e.g., via a website or repository).

Fig. 3. Collected data about non-technical aspects, from the survey.

6.2 Analysis of Macroprogramming Approaches

6.2.1 Paradigms. The distribution of macroprogramming languages across the approach clusters and the basic paradigms (cf. Section 4.4) is shown in Figure 4a and Figure 4b, respectively. Apparently, the majority of works follow a data-oriented approach; arguably, this reflects the fact that most of the works target the WSN domain, where the main goal is to extract data from the sensor network. This is also coherent with the fact that most of the macroprogramming languages take a declarative, specification stance. Beside that, all the main paradigms (logic, functional, imperative, object-oriented) have a discrete number of representatives—showing the orthogonality of the macro viewpoint to programming, as well as the consequence of embedding . On the other hand, only a handful of works take an ad-hoc approach that could not be framed as either control-, data-, space-time-, or ensemble-oriented.

6.2.2 Underlying platforms and languages. In Figure 4c, there is a hint about the underlying platforms or languages in which a given macroprogramming is supported or implemented. We denote with "*" that an approach supports multiple target platforms; with "None" that no implementation is provided or described; and we use a label "Other" to collect platforms for which only a single occurrence exists (this is the case, e.g., of embedded platforms, simulators, or individual languages such as Embedded Matlab, PeerSim, or Groovy, respectively). Several approaches found on the Java language and platform; various approaches target TinyOS, an operating system for WSN motes;

however, it is most frequent to address specific platforms (as a reflection of a rather wide coverage of target domains and paradigms).

6.2.3 Micro-level dependency. In Figure 4d, we observe that the majority of works do depend on micro-level entities (e.g., because they need to be addressed individually); however, there are also many works that abstract completely from the underlying set of components. Not surprisingly, only a few approaches are "scalable", allowing both to address individual nodes as well as to abstract from them entirely: these include Regiment [Newton et al. 2007] and aggregate programming [Viroli et al. 2019]. Indeed, the main goal of the surveyed macroprogramming approaches is to provide zero-cost abstractions to simplify the programming activity without renouncing to performance. Moreover, in several cases, the programming models aim to provide specific benefits: communication or execution efficiency (cf. WSN programming models), dynamic binding and architectures (cf. Scopes, DEECo, Warble, TECOLA), and self-organisation (cf. aggregate programming, Pleiades).

6.2.4 Macro-to-micro mapping. As it can be observed from Figure 4e, The implementation of macro-to-micro mapping is generally based on either *compilation* of the macro-program into the programs for the individual nodes (also known as *deglobalisation* or *global-to-local compilation*) or *interpretation* of the macroprogram according to some *execution protocol* (e.g., involving migratory execution of agents or orchestration of individuals). Quite frequent is also the approach based on *orchestration*, such as in Dolphin [Lima et al. 2018] or SmartSociety [Scekic et al. 2020], or the definition of additional entities like *mobile agents* which interact with micro-level entities to promote desired emergents, as in PIECES [Liu et al. 2003] or STOP/Chronus [Wada et al. 2010]. The less frequent mechanism is "context change", e.g., parameter setting as found in WOSP [Varughese et al. 2020] or market-based macroprogramming [Mainland et al. 2004], though these may not even be considered a "programming approach", strictly speaking.

Unfortunately, the macro-to-micro mapping is often not described formally (or even explicitly), which exceptions like SCEL [De Nicola et al. 2014], and aggregate programming [Beal et al. 2015]. For instance, in the latter approach, the core language – namely the field calculus – has a macro-level denotational semantics linked to the local operational semantics [Viroli et al. 2019], using computational fields (global data structures) as bridging abstraction.

6.2.5 Macroprogramming abstractions. Finally, we can observe that a number of abstractions or features recur in macroprogramming approaches. These include:

- *first-class groups*—the ability to directly express and manipulate groups of individuals (cf. group handles in EcoCast, swarms in Buzz or PaROS);
- *group lifecycle management*—the ability to evolve groups dynamically (cf. dynamic binding in Warble);
- *group addressing*—the ability to address a group, e.g., in terms of the individuals found in a certain spatial region or that share certain capabilities (cf. Regiment, SpatialViews, STOP/Chronus);
- *distributed state*—the ability to address the state of a group of stateful entities (cf. fields in aggregate programming, state rewriting in Comingle);
- *group inspection*—the ability to inspect or iterate over the individuals of a group (cf. node iteration in Kairos, resources in PyoT);
- *group goal decomposition*—the ability to consider a global goal and ways to split it across multiple individuals (cf. task partitioning in PaROS, spatial decomposition in Karma);
- *group communication*—the ability to get data from or push data to a group (cf. report/tell/-collective actions in makeSense, or neighbourhood-based communication in COSMOS, and aggregate programming);

- *information flow patterns*—the ability to specify how information should flow independently of structure or concrete communication mechanisms (cf. ATaG);
- *group-level actions*—the ability to express *what* a group should do (cf. functions in aggregate programming, activity nodes in DDFlow).

Sometimes, some of these aspects are abstracted away and implemented at the middleware layer: for instance, approaches that consider a WSN "like a database" let the programmer express a query (global goal) and then handle its partitioning into micro-actions through underlying mechanisms and execution protocols.

6.2.6 On implementation and abstracted concerns. A major goal of macroprogramming is abstracting from a series of low-level concerns. This is also strikingly evident from the quotes reported in Table 1, which suggest that the programmer can be relieved from "explicitly managing control, coordination, and state maintenance at the individual node level" [Bakshi and Prasanna 2005] in order to retrieve "simplicity and productivity" [Wada et al. 2008] through "focus on application specification rather than low-level details or inter-node messaging" [Awan et al. 2007]. Besides productivity, there is also the idea that low-level details can be addressed efficiently or opportunistically at the middleware level—see Section 6.3.2 for further considerations on this point.

The concerns that are abstracted may be classified according to the fundamental dimensions of structure, behaviour, and interaction. *Structural concerns* include connections between components and membership relationships. As these elements tend to change dynamically, expressing them in a declarative fashion enables the underlying platform to adopt flexible strategies for their reification. For instance, in Buzz [Pinciroli and Beltrame 2016], each robot follows a protocol to keep track of its membership in swarms, which further affects the set of its neighbours. In ScaFi [Casadei et al. 2021], groups self-organise by playing the logic expressed by the macroprogram in repeated sense-compute-interact rounds to continuously evaluate the "spatiotemporal boundary" of the process/ensemble. Therefore, we may conclude that often the macro-program is a piece of behaviour that is used to parametrise a larger behaviour, supported by a proper runtime system or middleware, which provides the "basic principle" for the collection of micro-level entities to act as a system.

Behavioural concerns that can be abstracted include specific decisions (e.g., what data must be stored or propagated), processing operations, and time aspects (e.g., when a certain behaviour is to be executed). For instance, in SNAP [Arashloo et al. 2016], the individual switches must determine how to route traffic and where the place state variables. As another example, macro-programs in aggregate computing [Beal et al. 2015], abstract from scheduling aspects, which enables dynamic tuning of the frequency at which devices operate, making time a "fluid" notion in such systems [Pianini et al. 2021b].

Interactional concerns are also often abstracted. In many cases, indeed, the details of communication, such as the specific format of the messages, the specific set of recipients, can be determined at runtime. Macroprogramming approaches for WSN, for instance, generally provide abstractions over routing and hop-by-hop information flows.

Among implementation strategies, a number of patterns recur The macroprogram can, as in PIECES [Liu et al. 2003] or STOP/Chronus [Wada et al. 2010], instruct mobile agents to move across the nodes of the network to access and process local state to infer global information. Orchestration – cf. Dolphin [Lima et al. 2018] and SmartSociety [Scekic et al. 2020] – is similar but does not involve moving agents. Related is also the approach, used for instance in Pyot [Azzara et al. 2014], based on inferring tasks from the macroprogram and distributing them over the set of micro-level entities. Round-based execution of macro-programs or projected micro-programs is also frequent and can be found both in asynchronous variants, as in aggregate computing [Beal et al. 2015], and in synchronous variants as in Giraph++ [Tian et al. 2013], SOSNA [Karpinski and Cahill 2008], and
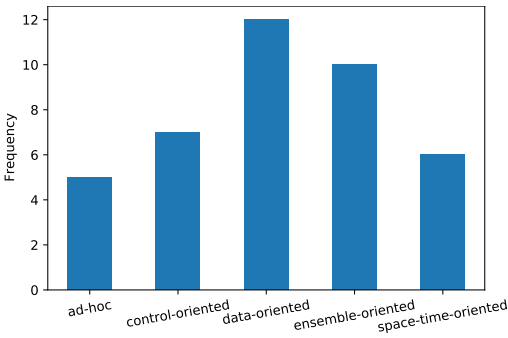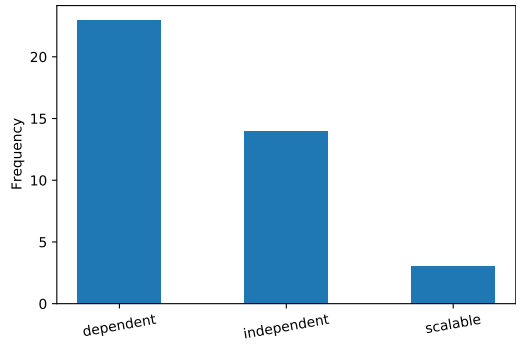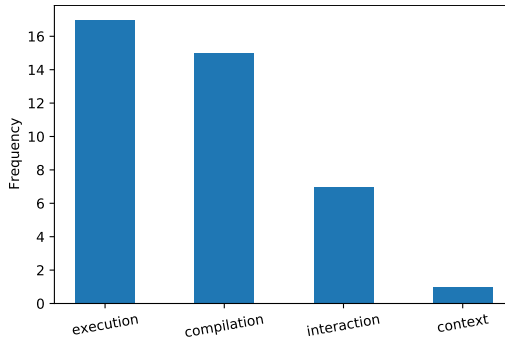
(a) Number of core works per approach.
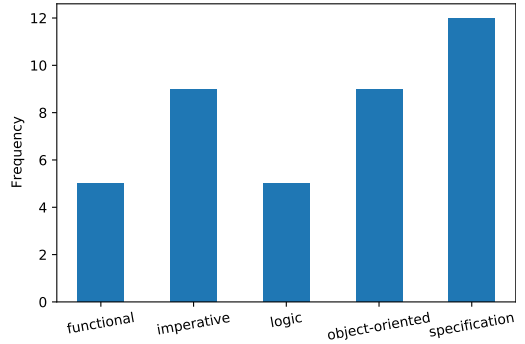
(b) Number of core works per paradigm.
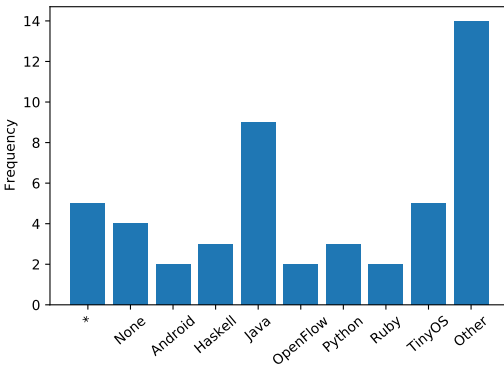
(c) Number of works per host platform/language. (*) means that multiple languages are supported.

(d) Number of core works per micro-level dependency level.

(e) Number of core works per macro-to-micro mapping approach.

Fig. 4. Collected data about technical aspects of the surveyed macroprogramming approaches.

WOSP [Varughese et al. 2020]. Such implementation strategies, beside "filling the abstraction gap", are also aimed at optimising application-specific concerns, which may include saving resources (e.g., energy or bandwidth) or promoting Quality of Service metrics like latency or reliability.

6.3 Opportunities

Research on macroprogramming provides opportunities (with corresponding challenges, covered in Section 6.4) in terms of synergies with related research fields and application domains.

6.3.1 Model- and Language-based Software Engineering. Models, as abstract representations of some aspect of a real or imagined object, play a key role in software engineering [Ludewig 2004]. Systems are generally described through multiple models covering different perspectives or viewpoints [Finkelstein et al. 1992]. As covered in Sections 3 to 5, *macroscopic perspectives* can be instrumental for directly addressing global properties, collective tasks, or system-level aspects. Indeed, we can observe that prominent perspectives in software modelling (e.g., structure, behaviour, and interaction) can be considered at a microscopic or a macroscopic level. For the latter:

- The *macro-structural view* considers the structural arrangement of multiple components of a system. The creation of macro-structures is sometimes a goal of macroprogramming (cf. topology programming in Pleiades [Bouget et al. 2018]).
- The *macro-behavioural view* considers behaviours emerging from multiple components of a system. This is generally the goal of macroprogramming: expressing the behaviour of a system as a whole (cf. swarm macroprogramming in Buzz [Pinciroli and Beltrame 2016]).
- The *macro-interactional view* considers interaction and communication at increasingly non-local levels. This is often instrumental to drive macro-behaviour by expressing how information flows among several components or across large structures (cf. collective communication interfaces in Abstract Regions [Welsh and Mainland 2004]).

Models have to be expressed in some language (also called a meta-model). Languages exist for specification, design, implementation, and verification of software, and contribute to a vision of *language-based software engineering* [Gupta 2015], which promotes the use of high-level DSLs for building software. Related notions such as *goal-oriented* [van Renesse 1998] or *declarative* programming [Baldoni et al. 2010; Lloyd 1994] are used to denote a similar idea: the use of languages to express an abstract model of a system emphasising *what* has to be achieved rather than *how*. The benefit is that the complexity for efficiently mapping the *what* to the *how* can be encapsulated in a middleware layer, while application developers can focus on domain abstractions and the business logic. In this sense, macroprogramming can be considered as a particular domain of declarative programming; however, we think that research in this field can potentially provide insights on the general principles and foundations of declarative programming.

6.3.2 Intelligent middlewares. Beside expressiveness, the abstraction provided by macroprogramming can foster the implementation of smart solutions at the middleware level. In early macroprogramming approaches on WSNs, the goal was often simplifying the programming activity (i.e., productivity) while keeping performance overhead at acceptable levels. In time, the idea of actually *improving* performance started to be considered as a research goal. Indeed, overfitting solutions may not be able to adequately generalise their performance to the various situations a system may experience in practice. On the other hand, more abstract architectures could adapt to diverse situations and do that *opportunistically*—by *proactively* looking for opportunities of optimisation. Of course, there is a trade-off between overfitting and underfitting models, and this revolves around a careful design of macroprogramming solutions in terms of (domain-specific) assumptions.

The middleware could be the part of the software system that implements the global-to-local mapping logic, possibly in a smart way. Such a smartness could serve to avoid unnecessary computations or communications, change structure to promote functionality, or re-configure the application to improve performance or resiliency. For instance, in aggregate computing [Casadei et al. 2020a] and MacroLab [Hnat et al. 2008], the logical macro-programmed system can be deployed variously

on available infrastructure, where different deployments may result in different non-functional trade-offs; moreover, their middleware can in principle adapt the deployment opportunistically as infrastructure, user preferences, or environmental conditions change. Indeed, a key opportunity would be to leverage recent advances in self-adaptive software and autonomic computing as well as artificial intelligence and machine learning.

6.3.3 Collective Intelligence, Soft Computing, Social Computing. There are several systems that are amenable to be studied and engineered by a collective perspective, as well as several research fields that address aspects of such collective systems [Tumer and Wolpert 2004]. Works on macro-programming are often found in such research areas (see Section 3 and Section 5.4), and might contribute (from its construction-oriented perspective) to the overall research endeavour about collective systems.

Computational collective intelligence (CCI) is a sub-field of AI that focusses on "the form of intelligence that emerges from the collaboration and competition of many individuals (artificial and/or natural)" [Szuba 2001]. The affinity with macroprogramming is evident, as the latter generally provides a means for expressing *what* collective intelligent behaviour should be like or work at, encapsulating the logic for building it in terms of rules of individual behaviour and interaction. However, the abstraction provided by strong macroprogramming languages tends to favour implementations achieving approximated solutions in complex situations. This is especially evident in macroprogramming languages for collective adaptive systems (Section 5.4), such as aggregate computing [Viroli et al. 2019], where macro-programs express global outcomes that are to be sought progressively in a self-organising fashion. In this sense, macroprogramming promotes a language-based approach to *soft computing* [Liang and He 2020], i.e. the use of computing to approximately solve very complex problems despite uncertainty, perturbations, and partial knowledge.

A recent systematic literature review on "collective intelligence" [Suran et al. 2020], covers conceptual frameworks and models for "collaborative problem solving and decision making", in the broad sense of *social computing* [Wang et al. 2007]—namely the paradigm where humans, society, and computing technology integrate to promote information representation, processing, communication, and use. The survey focusses on a high-level view and purposefully abstracts from specific domains—not even mentioned, the programming viewpoint is completely neglected. However, macroprogramming DSLs could work as inter-disciplinary artifacts capturing relationship and behaviour of groups and ecosystems. Benefits could be obtained by addressing issues at the right perspective.

6.4 Challenges

There are a number of challenges related to the engineering of macroprogramming systems. These include, e.g., designing macro-level abstractions, bridging macro-level abstractions with micro-level activity, formalising the macro-to-micro mapping, giving formal guarantees about the correctness of such a mapping, and integrating macroprogramming with more traditional approaches.

6.4.1 Abstraction and global-to-local mapping. A key challenge in macroprogramming is defining a good, coherent set of macro-level abstractions and identifying a proper way to map those to micro-level activity while promoting both functional and non-functional requirements. This also includes finding a balance between over-fitting and under-fitting solutions: the former may hinder reusability and extensibility, while the latter, as an attempt to achieve a one-size-fits-all support, may complicate implementations. As discussed previously, effective, highly-productive programming and smartness in implementations is where the most opportunities arise and arguably the major concerns for any macroprogramming language. The challenge revolves around ensuring that

global-to-locally mapped behaviour results, when actually carried out, in local-to-global effects in a consistent (and possibly efficient) way.

Moreover, some macroprogramming approaches such as, e.g., DEECo [Bures et al. 2013], SCEL [De Nicola et al. 2014], and aggregate programming [Beal et al. 2015], target *complex/collective adaptive systems* [Ferscha 2015]—see Section 5.4. Such systems feature complex networks of interactions that typically result in *emergent properties (emergents)* [Wolf and Holvoet 2004], namely macro-level properties that cannot be easily traced back to micro-level activity, because they are not – by definition – the result of mere summation of individual contributions (i.e., they are based on non-linear dynamics) [Holland 1998]. Due to its very nature, promoting desired emergents is a challenge. However, in some cases, emergence can be "steered". Existing research [Casadei et al. 2021] seems to suggest that macroprogramming may provide a privileged perspective and approach for steering emergent behaviour towards the desired emergents. In a sense, the development of a macroprogramming system might force its designers to approach the problem by a mixed top-down/bottom-up strategy.

6.4.2 Formal approaches to macroprogramming. In software engineering, the use of formal methods enables specification of non-ambiguous models of systems and promote their analysis and verification, possibly automated. In macroprogramming, languages backed by formal theories and calculi may be analysed to verify qualitative or quantitative properties. For instance, in aggregate programming it has been possible, by considering its core language (the field calculus), to prove Turing-like universality for space-time computations, identify language fragments supporting self-stabilising and density-independent computations, prove optimality theorems for specific algorithms or encodings, and promote deployment-independence at the middleware level [Viroli et al. 2019]. In SCEL [De Nicola et al. 2014], statistical model checking tools can be used to verify reachability properties, i.e., to compute the probability that a certain system configuration (e.g., expressed as a predicate on collected information) is reached within a certain deadline. Vice versa, several other macroprogramming languages focus mainly on providing a high-level API, simplifying the programming activity but providing little support for analysis and verification. In some cases, the semantics of the DSL is not even specified formally. Other approaches, such as WOSP [Varughese et al. 2020], provide certain properties (e.g., low communication overhead) by construction and use empirical methods (e.g., simulation) for verification. Therefore, a challenge related to the identification of good abstractions and global-to-local mapping strategies is the definition of formal frameworks supporting both correct and efficient implementations as well as discovery of properties and results (both at the application and middleware level). We note that this challenge (and opportunity) is also recognised by other fields of research including self-adaptive software and robotics threads [Farrell et al. 2018; Weyns et al. 2012].

Besides applying formal methods for verification and analysis within specific macroprogramming systems, another challenge lies in devising a *general, formal theory of macroprogramming* that abstracts from specific languages and possibly even from concrete paradigms. One possibility would be to rigorously identify a minimal but complete set of concepts or predicates applicable to programming systems to classify them as (a form of) macroprogramming. The basic principles provided in Section 4.2 could make for a starting point in this research. The use of such a formal framework could then be used to provide alternative, possibly more precise, classifications of macroprogramming approaches with respect to the one provided in Section 4.4.

6.4.3 Heterogeneity. A system is *heterogeneous* if it comprises different kinds of components. Macroprogramming a system of multiple heterogeneous components or individuals is challenging because making use of the different capabilities of these requires an individual-level viewpoint. Vice versa, macroprogramming homogeneous collectives (such as swarms of homogeneous robots) tends

to be simpler as any robot is assimilable to another. In principle, heterogeneity may be abstracted at the programming level and encapsulated at the middleware level, or code may be organised such that specific behaviour is modularised.

Moreover, heterogeneity is not only in shape or capabilities but also in aspects like autonomy and programmability. For instance, consider a heterogeneous cyber-physical collective made of smart city components (e.g., smart traffic lights, cloudlets, autonomous vehicles) and augmented human operators (e.g., through smartphones, smart watches or glasses), which may be programmed to support decentralised crowdsensing applications; the digital devices worn by those humans will move according to those humans' deliberation, and hence their mobility could not be programmed (but only "requested", at best). Among the surveyed approaches, only the SmartSociety platform [Scekic et al. 2020] provides some support for human orchestration, where humans and machines are considered *peers*.

While collectives tend to be homogeneous, heterogeneity is typically more present in *composites*, namely collections of entities related by a notion of *componenthood* [Masolo et al. 2020]. An example is a car, which builds on components such as engine, wheels, etc. However, it would be very hard to imagine the possibility of *programming* a car as a whole.

To conclude this reflection, macroprogramming does not need to assume homogeneity, but it does need to take heterogeneity into account at some level of its engineering stack (middleware, application, model). Moreover, we also observe that macroprogramming is not to be thought as a comprehensive approach meant to define all aspects of a system behaviour, which also leads to the following challenge.

6.4.4 Integration with other programming paradigms and toolchains. As discussed in previous sections, macroprogramming embodies a particular viewpoint of system development, which may not capture all the relevant functional and non-functional requirements. Indeed, a complex system may involve the solution of multiple different problems, each one best addressed by a specific paradigm. This is the idea of *multi-paradigm programming* [Albert et al. 2005; Spinellis et al. 1994]. On a more pragmatic side, supporting macroprogramming on top of existing development platforms (such as the JVM or .NET) may enable quick prototyping as well as reuse of features and tools from the host platform. This has fostered the emergence of *internal DSLs* [Voelter et al. 2013] for macroprogramming, which are embedded as expressive APIs on top of existing general-purpose languages: this is the case of PyoT (Python) [Azzara et al. 2014], Chronus (Ruby) [Wada et al. 2010], jDEECo (Java) [Bures et al. 2013], ScaFi for aggregate programming (Scala) [Casadei et al. 2020b], Dolphin (Groovy) [Lima et al. 2018], D'Artagnan (Haskell) [Mizzi et al. 2018], and AErlang (Erlang) [De Nicola et al. 2018]. However, this aspect of integration of paradigms poses *architectural* challenges, as macroprogramming tends to permeate various dimensions of the system—including structure, behaviour, and interaction. In summary, multi-paradigm programming is appealing but must be carefully analysed at the level of models, architecture, and development practice.

7 RELATED WORK

This work integrates, extends upon, and differentiates with respect to other survey papers. The main difference is that the secondary studies presented in the following, while similarly rich and detailed, adopt a narrower perspective (spatial computing, WSN, microelectromechanical systems, and swarm robotics, respectively). By contrast, this survey aims to relate various macroprogramming approaches across disparate domains, and adopts a general software engineering viewpoint. Moreover, due to their publication time, other surveys only cover works published before 2012. Indeed, by analysing the twenty-year time-frame from early 2000s to 2020, we can also make considerations about trends (see Section 3).

The most related survey is [Beal et al. 2012], which however focusses on *spatial computing* languages. It proposes a conceptual framework where spatial computation can be described in terms of constructs for *(i) measuring space-time* (sensors); *(ii) manipulating space-time* (actuators); *(iii) computation*; and *(iv) physical evolution* (inherent spatiotemporal dynamics). The device model accounts for the way devices are *discretised* in space-time (distinguishing between discrete, cellular, and continuous models), the way they are programmed (e.g., by giving them a uniform programs, heterogeneous programs, or leveraging mobile code), their communication scope (e.g., through local, neighbourhood, global regions), and their communication granularity (e.g., unicast, multicast, or broadcast). The survey classifies languages in the following groups: (i) amorphous computing (including pattern languages and manifold programming languages); (ii) biological modelling; (iii) agent-based modelling (including multi-agent and distributed systems modelling); (iv) wireless sensor networks (distinguishing between region-based, dataflow-based, database abstraction-based, centralised-view, and agent-based languages); (v) pervasive computing; (vi) swarm and modular robotics; (vii) parallel and reconfigurable computing (including dataflow, topological, and field languages); (viii) formal calculi for concurrency and distribution (i.e., process algebras/calculi). Languages are further analysed based on: characteristics of the language (type, DSL implementation pattern, platform, layers), supported spatial computing operators, and abstract device characteristics. Language type ranges over functional, imperative, declarative, graphical, process calculus, and any.

Very related is also [Mottola and Picco 2011], a 2011 survey that covers programming approaches for wireless sensor networks. In their taxonomy, the *interaction pattern* is classified into (i) *one-to-many*, (ii) *many-to-one*, and (iii) *many-to-many*. Moreover, the extent of distributed processing in space can be (i) *global*, e.g., in environment monitoring applications; or (i) *regional*, e.g., in intrusion detection or HVAC systems in buildings. Other dimensions include *goal* (sense-only or sense-and-react), *mobility* (static, mobile), *time* (periodic or event-driven). Regarding WSN programming abstractions, they define a taxonomy as follows. *Communication* aspects cover: *scope* (system-wide, physical neighbourhood-based, or multi-hop group); *addressing* (physical or logical); and *awareness* (implicit or explicit). *Computation* aspects include *scope* of computation (local, group, or global). The *model of data access* could be database, data sharing, mobile code, or message passing. Finally, the *paradigm* could be: *imperative* (sequential or event-driven); *declarative* (functional, rule-based, SQL-like, special-purpose); or *hybrid*.

The review [Brambilla et al. 2013] of swarm robotics from an engineering perspective neglects the programming viewpoint. However, they provide a taxonomy where collective behaviour is classified into behaviour for (i) *spatial organisation* (e.g., pattern formation, morphogenesis), (ii) *navigation and mobility* (e.g., coordinated motion and transport), (iii) *collective decision making* (e.g., consensus achievement and task allocation), and (iv) *other*. *Design methods* are categorised into *behaviour-based* (e.g., finite state machines, virtual physics-based) and *automatic* (e.g., evolutionary robotics and reinforcement learning-based methods). *Analysis methods* are categorised into *microscopic models*, *macroscopic models* (e.g., via rate/differential equations or control theory), and *real-robot analysis*.

Finally, certain works proposed concepts useful for classifying and understanding macroprogramming approaches. These elements have been considered and integrated into the taxonomy provided in Section 4.4. A possible classification of macroprogramming approaches [Choochaisri et al. 2012] distinguishes between

(1) *node-dependent* macroprogramming—where the nodes (or, more generally, the components of the micro-level) and their states are referred to explicitly by the macroprogram; and

(2) *node-independent* macroprogramming—where the underlying nodes are not visible at all to the programmer.

As per the discussion of Section 4.3, node-dependent approaches tend to enact a weak form of macroprogramming. Examples of node-independent approaches include, e.g., those that abstract a WSN as a database. Another distinction can be made between:

(1) *data-driven* macroprogramming [Pathak and Prasanna 2010]—where macro-programs define tasks consuming and producing data; and

(2) *control-driven* macroprogramming [Bakshi and Prasanna 2005]—where macro-programs specify control flow and instructions operating on distributed memory.

The classification in data-driven and control-driven approaches has been applied in other fields such as coordination [Papadopoulos and Arbab 1998], where the latter are also known as *task-* or *process-oriented* coordination models.

8 CONCLUSION

For the first time, we provide an explicit and integrated view of research on macroprogramming—the paradigm aimed at expressing and executing the global behaviour of systems of computational entities. The manuscript discusses what macroprogramming is *per se*, its core application domains, its main concepts, and analyses and classifies a wide range of works addressing system development by a more-or-less macroscopic perspective. Thus, it provides a more general, comprehensive, and up-to-date coverage of macroprogramming with respect to previous works, which covered it in the context of engineering approaches for wireless sensor networks [Mottola and Picco 2011], spatial computing systems [Beal et al. 2012], and swarm robotics [Brambilla et al. 2013].

We argue that a macro-level stance could be beneficial for software engineering especially in forthcoming distributed computing scenarios (cf. swarm robotics, large-scale CPSs, the IoT, and smart cities), and for promoting language-based solutions to collective adaptive behaviour and intelligence [De Nicola et al. 2020]. Indeed, for the *collective computing revolution* [Abowd 2016] to fully unfold, there will be needed tools to harness the complexity of large ecosystems involving machines as well as humans [Hendler and Berners-Lee 2010]. In particular, the macro-level perspective could represent a *complementary viewpoint* for addressing structure, behaviour, and interaction in complex socio-technical systems. However, macroprogramming comes with peculiar challenges, at the border of science and engineering, such as those related to "steering emergent behaviour" (i.e., promoting desired emergents while avoiding undesired emergents [Schmeck 2005]), "guiding self-organisation" [Prokopenko 2014], promoting collective intelligence [Suran et al. 2020], and, in general, formally expressing global/system-level intents, and mapping those to micro-level instructions—possibly with guarantees.

We suggest that macroprogramming can be considered as an *abstract paradigm* (e.g., similarly to the notion of declarative programming), for it conveys a distinguishing perspective to programming and a coherent set of principles (cf. Section 4). Then, concrete macroprogramming languages can adopt specific programming paradigms (e.g., imperative, functional, logic, or object-oriented), approaches (e.g., control-, data-, space-time-, and ensemble-oriented), and mechanisms (e.g., first-class groups, collective communication interfaces, distributed state/data structures, etc.). Macroprogramming languages tend to be domain-specific (e.g., addressing data collection and transformation in WSANs, or behaviour and actuation in robot swarms), since domain assumptions are generally instrumental to properly and efficiently map high-level abstractions to activity on the low-level platform. However, there is arguably margin for recovering *general principles* through inter-domain discussion and sharing of ideas, but this would require a more integrated and structured view of macroprogramming as a field, which this article aims to cultivate.

REFERENCES

2007. *Computing Media and Languages for Space-Oriented Computation, 03.09. - 08.09.2006.* Dagstuhl Seminar Proceedings, Vol. 06361. http://drops.dagstuhl.de/portals/06361/

Gregory D. Abowd. 2016. Beyond Weiser: From Ubiquitous to Collective Computing. *Computer* 49, 1 (2016), 17–23. https://doi.org/10.1109/MC.2016.22

Julie A. Adams. 2001. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. *AI Mag.* 22, 2 (2001), 105–108. https://doi.org/10.1609/aimag.v22i2.1567

Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver, and Germán Vidal. 2005. Operational semantics for declarative multi-paradigm languages. *J. Symb. Comput.* 40, 1 (2005), 795–829. https://doi.org/10.1016/j.jsc.2004.01.001

J.C. Alexander. 1987. *The Micro-macro Link.* University of California Press. https://books.google.it/books?id=ClWF4cw5qc4C

Yehia Abd Alrahman, Rocco De Nicola, Michele Loreti, Francesco Tiezzi, and Roberto Vigo. 2015. A calculus for attribute-based communication. In *Proceedings of the 30th ACM Symposium on Applied Computing (SAC'15).* ACM, 1840–1845. https://doi.org/10.1145/2695664.2695668

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetkAT: semantic foundations for networks. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14).* ACM, 113–126. https://doi.org/10.1145/2535838.2535862

William F. Appelbe and Klaus Hansen. 1985. A Survey of Systems Programming Languages: Concepts and Facilities. *Softw. Pract. Exp.* 15, 2 (1985), 169–190. https://doi.org/10.1002/spe.4380150205

Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *Proceedings of the ACM SIGCOMM 2016 Conference.* ACM, 29–43. https://doi.org/10.1145/2934872.2934892

Michael P. Ashley-Rollman, Seth Copen Goldstein, Peter Lee, Todd C. Mowry, and Padmanabhan Pillai. 2007. Meld: A declarative approach to programming ensembles. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems.* IEEE, 2794–2800. https://doi.org/10.1109/IROS.2007.4399480

Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A survey. *Comput. Networks* 54, 15 (2010), 2787–2805. https://doi.org/10.1016/j.comnet.2010.05.010

Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. 2017. Compositional Blocks for Optimal Self-Healing Gradients. In *11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017, Tucson, AZ, USA, September 18-22, 2017.* IEEE Computer Society, 91–100. https://doi.org/10.1109/SASO.2017.18

Asad Awan, Suresh Jagannathan, and Ananth Grama. 2007. Macroprogramming heterogeneous sensor networks using COSMOS. In *Proceedings of the 2007 EuroSys Conference.* ACM, 159–172. https://doi.org/10.1145/1272996.1273014

Andrea Azzara, Daniele Alessandrelli, Stefano Bocchino, Matteo Petracca, and Paolo Pagano. 2014. PyoT, a macroprogramming framework for the Internet of Things. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems, SIES 2014, Pisa, Italy, June 18-20, 2014.* IEEE, 96–103. https://doi.org/10.1109/SIES.2014.6871193

Jonathan Bachrach, Jacob Beal, and James McLurkin. 2010. Composable continuous-space programs for robotic swarms. *Neural Comput. Appl.* 19, 6 (2010), 825–847. https://doi.org/10.1007/s00521-010-0382-8

Jos C. M. Baeten. 2005. A brief history of process algebra. *Theor. Comput. Sci.* 335, 2-3 (2005), 131–146. https://doi.org/10.1016/j.tcs.2004.07.036

Amol Bakshi and Viktor K. Prasanna. 2005. Programming Paradigms for Networked Sensing: A Distributed Systems' Perspective. In *7th International Workshop on Distributed Computing (IWDC'05), Proceedings (LNCS)*, Vol. 3741. Springer, 451–462. https://doi.org/10.1007/11603771_50

Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. 2013. The Abstract Task Graph: A Methodology for Architecture-Independent Programming of Networked Sensor Systems. In *Workshop on End-to-End, Sense-and-Respond Systems, Applications, and Services (EESR '05).* ACM / USENIX. https://doi.org/10.5555/1072530.1072535

Matteo Baldoni, Cristina Baroglio, Viviana Mascardi, Andrea Omicini, and Paolo Torroni. 2010. Agents, Multi-Agent Systems and Declarative Programming: What, When, Where, Why, Who, How? In *A 25-Year Perspective on Logic Programming: Achievements of the Italian Association for Logic Programming.* LNCS, Vol. 6125. Springer, 204–230. https://doi.org/10.1007/978-3-642-14309-0_10

Jacob Beal and Jonathan Bachrach. 2006. Infrastructure for Engineered Emergence in Sensor/Actuator Networks. *IEEE Intelligent Systems* 21 (2006), 10–19. Issue 2. https://doi.org/10.1109/MIS.2006.29

Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. 2012. Organizing the Aggregate: Languages for Spatial Computing. *CoRR* abs/1202.5509 (2012). arXiv:1202.5509 http://arxiv.org/abs/1202.5509

Jacob Beal, Danilo Pianini, and Mirko Viroli. 2015. Aggregate Programming for the Internet of Things. *Computer* 48, 9 (2015), 22–30. https://doi.org/10.1109/MC.2015.261

Ryan Beckett, Ratul Mahajan, Todd D. Millstein, Jitendra Padhye, and David Walker. 2019. Don't mind the gap: Bridging network-wide objectives and device-level configurations: brief reflections on abstractions for network programming. *Comput. Commun. Rev.* 49, 5 (2019), 104–106. https://doi.org/10.1145/3371934.3371965

Olivier Boissier, Rafael H. Bordini, Jomi Fred Hübner, Alessandro Ricci, and Andrea Santi. 2013. Multi-agent oriented programming with JaCaMo. *Sci. Comput. Program.* 78, 6 (2013), 747–761. https://doi.org/10.1016/j.scico.2011.10.004

Simon Bouget, Yérom-David Bromberg, Adrien Luxey, and François Taïani. 2018. Pleiades: Distributed Structural Invariants at Scale. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018*. IEEE Computer Society, 542–553. https://doi.org/10.1109/DSN.2018.00062

Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. 2013. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intell.* 7, 1 (2013), 1–41. https://doi.org/10.1007/s11721-012-0075-2

Tomás Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. 2013. DEECO: an ensemble-based component system. In *CBSE'13, Proceedings of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering*. ACM, 81–90. https://doi.org/10.1145/2465449.2465462

Roberto Casadei, Danilo Pianini, Andrea Placuzzi, Mirko Viroli, and Danny Weyns. 2020a. Pulverization in Cyber-Physical Systems: Engineering the Self-Organizing Logic Separated from Deployment. *Future Internet* 12, 11 (2020), 203. https://doi.org/10.3390/fi12110203

Roberto Casadei, Mirko Viroli, Giorgio Audrito, and Ferruccio Damiani. 2020b. FScaFi : A Core Calculus for Collective Adaptive Systems Programming. In *9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Proceedings, Part II (LNCS)*, Vol. 12477. Springer, 344–360. https://doi.org/10.1007/978-3-030-61470-6_21

Roberto Casadei, Mirko Viroli, Giorgio Audrito, Danilo Pianini, and Ferruccio Damiani. 2021. Engineering collective intelligence at the edge with aggregate processes. *Eng. Appl. Artif. Intell.* 97 (2021), 104081. https://doi.org/10.1016/j.engappai.2020.104081

Supasate Choochaisri, Nuttanart Pornprasitsakul, and Chalermek Intanagonwiwat. 2012. Logic Macroprogramming for Wireless Sensor Networks. *Int. J. Distributed Sens. Networks* 8 (2012). https://doi.org/10.1155/2012/171738

Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley R. Schmerl, and Gabriel Tamura et al. 2010. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In *Software Engineering for Self-Adaptive Systems II (LNCS)*, Vol. 7475. Springer, 1–32. https://doi.org/10.1007/978-3-642-35813-5_1

Rocco De Nicola, Tan Duong, Omar Inverso, and Catia Trubiani. 2018. AErlang: Empowering Erlang with attribute-based communication. *Sci. Comput. Program.* 168 (2018), 71–93. https://doi.org/10.1016/j.scico.2018.08.006

Rocco De Nicola, Stefan Jähnichen, and Martin Wirsing. 2020. Rigorous engineering of collective adaptive systems: special section. *Int. J. Softw. Tools Technol. Transf.* 22, 4 (2020), 389–397. https://doi.org/10.1007/s10009-020-00565-0

Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. 2014. A Formal Approach to Autonomic Systems Programming: The SCEL Language. *ACM Trans. Auton. Adapt. Syst.* 9, 2 (2014), 7:1–7:29. https://doi.org/10.1145/2619998

Dimitris Dedousis and Vana Kalogeraki. 2018. A Framework for Programming a Swarm of UAVs. In *11th PErvasive Technologies Related to Assistive Environments Conference (PETRA'18), Proceedings*. ACM, 5–12. https://doi.org/10.1145/3197768.3197772

Naranker Dulay, Matteo Micheletti, Leonardo Mostarda, and Andrea Piermarteri. 2018. PICO-MP: De-centralised Macro-Programming for Wireless Sensor and Actuator Networks. In *32nd IEEE International Conference on Advanced Information Networking and Applications, AINA 2018*. IEEE Comp. Soc., 289–296. https://doi.org/10.1109/AINA.2018.00052

Marie Farrell, Matt Luckcuck, and Michael Fisher. 2018. Robotics and Integrated Formal Methods: Necessity Meets Opportunity. In *Integrated Formal Methods - 14th International Conference, IFM 2018, Proceedings (Lecture Notes in Computer Science)*, Vol. 11023. Springer, 161–171. https://doi.org/10.1007/978-3-319-98938-9_10

Alois Ferscha. 2015. Collective Adaptive Systems. In *2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing, Proceedings of (UbiComp/ISWC'15 Adjunct)*. ACM, 893–895. https://doi.org/10.1145/2800835.2809508

Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, and Benedict du Boulay et al. 2020. Capturing and Characterising Notional Machines. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2020*. ACM, 502–503. https://doi.org/10.1145/3341525.3394988

Anthony Finkelstein, Jeff Kramer, Bashar Nuseibeh, L. Finkelstein, and Michael Goedicke. 1992. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *Int. J. Softw. Eng. Knowl. Eng.* 2, 1 (1992), 31–57. https://doi.org/10.1142/S0218194092000038

Stacy E. Finkelstein, Peter J. Freyd, and James Lipton. 2003. A new framework for declarative programming. *Theor. Comput. Sci.* 300, 1-3 (2003), 91–160. https://doi.org/10.1016/S0304-3975(01)00308-5

Robert L Flood. 1994. I keep six honest serving men: they taught me all I knew. *System Dynamics Review* 10, 2-3 (1994), 231–243.

Jacques Gignoux, Guillaume Chérel, Ian D Davies, Shayne R Flint, and Eric Lateltin. 2017. Emergence and complex systems: The contribution of dynamic graph theory. *Ecological Complexity* 31 (2017), 34–49. https://doi.org/10.1016/j.ecocom.2017.02.006

Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, and Martín Casado et al. 2008. NOX: towards an operating system for networks. *Comput. Commun. Rev.* 38, 3 (2008), 105–110. https://doi.org/10.1145/1384609.1384625

Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. 2005. Macro-programming Wireless Sensor Networks Using *Kairos*. In *Distributed Computing in Sensor Systems, 1st IEEE Int. Conf. , DCOSS 2005 (LNCS)*, Vol. 3560. Springer,

126–140. https://doi.org/10.1007/11502593_12

Gopal Gupta. 2015. Language-based software engineering. *Sci. Comput. Program.* 97 (2015), 37–40. https://doi.org/10.1016/j.scico.2014.02.010

Vikram Gupta, Junsung Kim, Aditi Pandya, Karthik Lakshmanan, Ragunathan Rajkumar, and Eduardo Tovar. 2011. Nano-CF: A coordination framework for macro-programming in Wireless Sensor Networks. In *8th Annual IEEE Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON'11), Proceedings*. IEEE, 467–475. https://doi.org/10.1109/SAHCN.2011.5984931

Heiko Hamann. 2010. *Space-Time Continuous Models of Swarm Robotic Systems - Supporting Global-to-Local Programming*. Cognitive Systems Monographs, Vol. 9. Springer. https://doi.org/10.1007/978-3-642-13377-0

Jim Hendler and Tim Berners-Lee. 2010. From the Semantic Web to social machines: A research challenge for AI on the World Wide Web. *Artif. Intell.* 174, 2 (2010), 156–161. https://doi.org/10.1016/j.artint.2009.11.010

Timothy W. Hnat, Tamim I. Sookoor, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. 2008. MacroLab: A Vector-Based Macroprogramming Framework for Cyber-Physical Systems. In *6th ACM Conf. on Embedded Network Sensor Systems, Proceedings of*. ACM, 225–238. https://doi.org/10.1145/1460412.1460435

J.H. Holland. 1998. *Emergence: From Chaos to Order*. Oxford University Press.

Clare Horsman, Susan Stepney, Robert C. Wagner, and Viv Kendon. 2013. When does a physical system compute? *CoRR* abs/1309.7979 (2013). arXiv:1309.7979 http://arxiv.org/abs/1309.7979

Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. 2007. Developing organised multiagent systems using the MOISE$^+$ model: programming issues at the system and agent levels. *Int. J. Agent Oriented Softw. Eng.* 1, 3/4 (2007), 370–395. https://doi.org/10.1504/IJAOSE.2007.016266

Daniel Jacobi, Pablo E Guerrero, Ilia Petrov, and Alejandro Buchmann. 2008. Structuring sensor networks with scopes. In *3rd IEEE European Conference on Smart Sensing and Context (EuroSSC), IEEE Communications Society, Zurich*.

Yaochu Jin and Yan Meng. 2011. Morphogenetic Robotics: An Emerging New Field in Developmental Robotics. *IEEE Trans. Syst. Man Cybern. Part C* 41, 2 (2011), 145–160. https://doi.org/10.1109/TSMCC.2010.2057424

Somayeh Kalantari, Eslam Nazemi, and Behrooz Masoumi. 2020. Emergence phenomena in self-organizing systems: a systematic literature review of concepts, researches, and future prospects. *J. Organ. Comput. Electron. Commer.* 30, 3 (2020), 224–265. https://doi.org/10.1080/10919392.2020.1748977

Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. 2013. Optimizing the "one big switch" abstraction in software-defined networks. In *Conference on emerging Networking Experiments and Technologies (CoNEXT'13), Proceedings of*. ACM, 13–24. https://doi.org/10.1145/2535372.2535373

Marcin Karpinski and Vinny Cahill. 2008. Stream-based macro-programming of wireless sensor, actuator network applications with SOSNA. In *Proceedings of the 5th Workshop on Data Management for Sensor Networks*. ACM, 49–55. https://doi.org/10.1145/1402050.1402061

James Kennedy. 2006. Swarm Intelligence. In *Handbook of Nature-Inspired and Innovative Computing - Integrating Classical Models with Emerging Technologies*. Springer, 187–219. https://doi.org/10.1007/0-387-27705-6_6

Jeffrey O. Kephart and David M. Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

Gregor Kiczales. 1992. Towards a new model of abstraction in the engineering of software. In *International Workshop on Reflection and Meta-Level Architecture*. Citeseer, 67–76.

Rudyard Kipling. 1902. I keep six honest serving men. *Just so stories* (1902).

Barbara Ann Kitchenham and Stuart Charters. 2007. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE 2007-001. Keele University and Durham University Joint Report.

Manos Koutsoubelias and Spyros Lalis. 2016. TeCoLa: A Programming Framework for Dynamic and Heterogeneous Robotic Teams. In *Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2016)*. ACM, 115–124. https://doi.org/10.1145/2994374.2994397

Diego Kreutz, Fernando M. V. Ramos, Paulo Jorge Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (2015), 14–76. https://doi.org/10.1109/JPROC.2014.2371999

Shriram Krishnamurthi and Kathi Fisler. 2019. Programming paradigms and beyond. *The Cambridge Handbook of Computing Education Research* 37 (2019).

Edmund Soon Lee Lam, Iliano Cervesato, and Nabeeha Fatima. 2015. Comingle: Distributed Logic Programming for Decentralized Mobile Ensembles. In *Coordination Models and Languages - 17th IFIP WG 6.1 International Conference, COORDINATION 2015 (LNCS)*, Vol. 9037. Springer, 51–66. https://doi.org/10.1007/978-3-319-19282-6_4

Junbin Liang, Jiannong Cao, Rui Liu, and Tao Li. 2016. Distributed Intelligent MEMS: A Survey and a Real-Time Programming Framework. *ACM Comput. Surv.* 49, 1 (2016), 20:1–20:29. https://doi.org/10.1145/2926964

Yun Liang and Tian-ping He. 2020. Survey on soft computing. *Soft Comput.* 24, 2 (2020), 761–770. https://doi.org/10.1007/s00500-019-04508-z

Yannis Lilis and Anthony Savidis. 2020. A Survey of Metaprogramming Languages. *ACM Comput. Surv.* 52, 6 (2020), 113:1–113:39. https://doi.org/10.1145/3354584

Aliandro Lima, Walfredo Cirne, Francisco Vilar Brasileiro, and Daniel Fireman. 2006. A Case for Event-Driven Distributed Objects. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE 2006. Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 4276. Springer, 1705–1721. https://doi.org/10.1007/11914952_46

Keila Lima, Eduardo R. B. Marques, José Pinto, and João B. Sousa. 2018. Dolphin: A Task Orchestration Language for Autonomous Vehicle Networks. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2018, Madrid, Spain, October 1-5, 2018*. IEEE, 603–610. https://doi.org/10.1109/IROS.2018.8594059

Jessica Lindblom and Tom Ziemke. 2003. Social Situatedness of Natural and Artificial Intelligence: Vygotsky and Beyond. *Adapt. Behav.* 11, 2 (2003), 79–96. https://doi.org/10.1177/10597123030112002

Jie Liu, Maurice Chu, Juan Liu, James Reich, and Feng Zhao. 2003. State-Centric Programming for Sensor-Actuator Network Systems. *IEEE Pervasive Comput.* 2, 4 (2003), 50–62. https://doi.org/10.1109/MPRV.2003.1251169

John W. Lloyd. 1994. Practical Advantages of Declarative Programming. In *1994 Joint Conference on Declarative Programming, GULP-PRODE 1994, Volume 1*. 18–30.

Michele Loreti and Jane Hillston. 2016. Modelling and Analysis of Collective Adaptive Systems with CARMA and its Tools. In *Formal Methods for the Quantitative Evaluation of Collective Adaptive (LNCS)*, Vol. 9700. Springer, 83–119. https://doi.org/10.1007/978-3-319-34096-8_4

Jochen Ludewig. 2004. Models in software engineering - an introduction. *Inform. Forsch. Entwickl.* 18, 3-4 (2004), 105–112. https://doi.org/10.1007/s00450-004-0155-7

Samuel Madden, Robert Szewczyk, Michael J. Franklin, and David E. Culler. 2002. Supporting Aggregate Queries Over Ad-Hoc Wireless Sensor Networks. In *4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002)*. IEEE Computer Society, 49–58. https://doi.org/10.1109/MCSA.2002.1017485

Geoffrey Mainland, Laura Kang, Sébastien Lahaie, David C. Parkes, and Matt Welsh. 2004. Using virtual markets to program global behavior in sensor networks. In *Proceedings of the 11st ACM SIGOPS European Workshop*. ACM, 1. https://doi.org/10.1145/1133572.1133587

Geoffrey Mainland, Greg Morrisett, and Matt Welsh. 2008. Flask: staged functional programming for sensor networks. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*. ACM, 335–346. https://doi.org/10.1145/1411204.1411251

Marco Mamei. 2011. Macro Programming a Spatial Computer with Bayesian Networks. *ACM Trans. Auton. Adapt. Syst.* 6, 2 (2011), 16:1–16:25. https://doi.org/10.1145/1968513.1968519

Marco Mamei, Franco Zambonelli, and Letizia Leonardi. 2004. Co-Fields: A Physically Inspired Approach to Motion Coordination. *IEEE Pervasive Comput.* 3, 2 (2004), 52–61. https://doi.org/10.1109/MPRV.2004.1316820

Pedro MN Martins and Julie A McCann. 2017. Network-wide programming challenges in cyber-physical systems. In *Cyber-physical systems*. Elsevier, 103–113. https://doi.org/10.1016/B978-0-12-803801-7.00007-9

Claudio Masolo, Laure Vieu, Roberta Ferrario, Stefano Borgo, and Daniele Porello. 2020. Pluralities, Collectives, and Composites. In *Formal Ontology in Information Systems - Proceedings of the 11th International Conference (FOIS 2020) (Frontiers in Artificial Intelligence and Applications)*, Vol. 330. IOS Press, 186–200. https://doi.org/10.3233/FAIA200671

Adrian Mizzi, Joshua Ellul, and Gordon J. Pace. 2018. D'Artagnan: An Embedded DSL Framework for Distributed Embedded Systems. In *Proceedings of the Real World Domain Specific Languages Workshop, RWDSL@CGO 2018, Vienna, Austria, February 24-24, 2018*. ACM, 2:1–2:9. https://doi.org/10.1145/3183895.3183899

Adrian Mizzi, Joshua Ellul, and Gordon J. Pace. 2019. Porthos: Macroprogramming Blockchain Systems. In *10th IFIP International Conference on New Technologies, Mobility and Security (NTMS 2019)*. IEEE, 1–5. https://doi.org/10.1109/NTMS.2019.8763784

George E. Mobus and Michael C. Kalton. 2014. *Principles of Systems Science*. Springer Publishing Company, Incorporated.

Kief Morris. 2016. *Infrastructure as Code: Managing Servers in the Cloud* (1st ed.). O'Reilly Media, Inc.

Luca Mottola, Mattia Moretta, Kamin Whitehouse, and Carlo Ghezzi. 2014. Team-level programming of drone sensor networks. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys'14)*. ACM, 177–190. https://doi.org/10.1145/2668332.2668353

Luca Mottola and Gian Pietro Picco. 2011. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.* 43, 3 (2011), 19:1–19:51. https://doi.org/10.1145/1922649.1922656

Luca Mottola, Gian Pietro Picco, Felix Jonathan Oppermann, Joakim Eriksson, Niclas Finne, and Harald Fuchs et al. 2019. makeSense: Simplifying the Integration of Wireless Sensor Networks into Business Processes. *IEEE Trans. Software Eng.* 45, 6 (2019), 576–596. https://doi.org/10.1109/TSE.2017.2787585

Ryan Newton, Greg Morrisett, and Matt Welsh. 2007. The regiment macroprogramming system. In *Proceedings of the 6th Int. Conf. on Information Processing in Sensor Networks, IPSN 2007*. ACM, 489–498. https://doi.org/10.1145/1236360.1236422

Ryan Newton and Matt Welsh. 2004. Region streams: functional macroprogramming for sensor networks. In *Proceedings of the 1st Workshop on Data Management for Sensor Networks, DMSN 2004 (ACM International Conference Proceeding Series)*,

Vol. 72. ACM, 78–87. https://doi.org/10.1145/1052199.1052213

Yang Ni, Ulrich Kremer, Adrian Stere, and Liviu Iftode. 2005. Programming ad-hoc networks of mobile and resource-constrained devices. (2005), 249–260. https://doi.org/10.1145/1065010.1065040

Joseph Noor, Hsiao-Yun Tseng, Luis Garcia, and Mani B. Srivastava. 2019. DDFlow: visualized declarative programming for heterogeneous IoT networks. In *Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI 2019*. ACM, 172–177. https://doi.org/10.1145/3302505.3310079

Kristen Nygaard. 1997. GOODS to Appear on the Stage. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Proceedings (LNCS)*, Vol. 1241. Springer, 1–31. https://doi.org/10.1007/BFb0053372

George A. Papadopoulos and Farhad Arbab. 1998. Coordination Models and Languages. *Adv. Comput.* 46 (1998), 329–400. https://doi.org/10.1016/S0065-2458(08)60208-9

Animesh Pathak and Viktor K. Prasanna. 2010. Energy-Efficient Task Mapping for Data-Driven Sensor Network Macroprogramming. *IEEE Trans. Computers* 59, 7 (2010), 955–968. https://doi.org/10.1109/TC.2009.168

Animesh Pathak and Viktor K. Prasanna. 2011. High-Level Application Development for Sensor Networks: Data-Driven Approach. In *Theoretical Aspects of Distributed Computing in Sensor Networks*. Springer, 865–891. https://doi.org/10.1007/978-3-642-14849-1_26

Danilo Pianini, Roberto Casadei, Mirko Viroli, Stefano Mariani, and Franco Zambonelli. 2021b. Time-Fluid Field-Based Coordination through Programmable Distributed Schedulers. *Logical Methods in Computer Science* Volume 17, Issue 4 (Nov. 2021). https://doi.org/10.46298/lmcs-17(4:13)2021

Danilo Pianini, Roberto Casadei, Mirko Viroli, and Antonio Natali. 2021a. Partitioned integration and coordination via the self-organising coordination regions pattern. *Future Gener. Comput. Syst.* 114 (2021), 44–68. https://doi.org/10.1016/j.future.2020.07.032

Danilo Pianini, Mirko Viroli, and Jacob Beal. 2015. Protelis: practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC'15*. ACM, 1846–1853. https://doi.org/10.1145/2695664.2695913

Carlo Pinciroli and Giovanni Beltrame. 2016. Buzz: A Programming Language for Robot Swarms. *IEEE Softw.* 33, 4, 97–100. https://doi.org/10.1109/MS.2016.95

Mikhail Prokopenko (Ed.). 2014. *Guided Self-Organization: Inception.* Springer. https://doi.org/10.1007/978-3-642-53734-9

Yuansong Qiao, Robert Nolani, Saul Gill, Guiming Fang, and Brian Lee. 2018. ThingNet: A micro-service based IoT macroprogramming platform over edges and cloud. In *21st Conference on Innovation in Clouds, Internet and Networks and Workshops, ICIN 2018, Paris, France, February 19-22, 2018*. IEEE, 1–4. https://doi.org/10.1109/ICIN.2018.8401626

Adil Rasheed, Omer San, and Trond Kvamsdal. 2020. Digital Twin: Values, Challenges and Enablers From a Modeling Perspective. *IEEE Access* 8 (2020), 21980–22012. https://doi.org/10.1109/ACCESS.2020.2970143

Yosef Saputra, Jie Hua, Nathaniel Wendt, Christine Julien, and Gruia-Catalin Roman. 2019. Warble: programming abstractions for personalizing interactions in the internet of things. In *Proceedings of the 6th Int. Conf. on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2019*. IEEE/ACM, 128–139. https://doi.org/10.1109/MOBILESoft.2019.00026

Ognjen Scekic, Tommaso Schiavinotto, Svetoslav Videnov, Michael Rovatsos, Hong Linh Truong, Daniele Miorandi, and Schahram Dustdar. 2020. A Programming Model for Hybrid Collaborative Adaptive Systems. *IEEE Trans. Emerg. Top. Comput.* 8, 1 (2020), 6–19. https://doi.org/10.1109/TETC.2017.2702578

Michael Schillo, Klaus Fischer, and Christof T. Klein. 2000. The Micro-Macro Link in DAI and Sociology. In *Multi-Agent-Based Simulation, 2nd Int. Workshop, MABS 2000 (LNCS)*, Vol. 1979. Springer, 133–148. https://doi.org/10.1007/3-540-44561-7_10

Hartmut Schmeck. 2005. Organic Computing - A New Vision for Distributed Embedded Systems. In *8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005)*. IEEE Computer Society, 201–203. https://doi.org/10.1109/ISORC.2005.42

Dimitrios Serpanos. 2018. The Cyber-Physical Systems Revolution. *Computer* 51, 3 (2018), 70–73. https://doi.org/10.1109/MC.2018.1731058

Tamim I Sookoor. 2009. *The Design of MDB: a Macrodebugger for Wireless Embedded Networks.* Ph.D. Dissertation. University of Virginia.

Diomidis Spinellis, Sophia Drossopoulou, and Susan Eisenbach. 1994. Language and Architecture Paradigms as Object Classes. In *Programming Languages and System Architectures, Int. Conf., Proceedings (LNCS)*, Vol. 782. Springer, 191–207. https://doi.org/10.1007/3-540-57840-4_32

Joel Spolsky. 2004. The law of leaky abstractions. In *Joel on Software*. Springer, 197–202.

Shweta Suran, Vishwajeet Pattanaik, and Dirk Draheim. 2020. Frameworks for Collective Intelligence: A Systematic Literature Review. *ACM Comput. Surv.* 53, 1 (2020), 14:1–14:36. https://doi.org/10.1145/3368986

T.M. Szuba. 2001. *Computational Collective Intelligence.* Wiley.

Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think Like a Vertex" to "Think Like a Graph". *Proc. VLDB Endow.* 7, 3 (2013), 193–204. https://doi.org/10.14778/2732232.2732238

Yi-Hsuan Tu, Yen-Chiu Lee, Yi-Wei Tsai, Pai H. Chou, and Ting-Chou Chien. 2011. EcoCast: Interactive, object-oriented macroprogramming for networks of ultra-compact wireless sensor nodes. In *10th International Conference on Information*

Processing in Sensor Networks (IPSN'11), Proceedings. IEEE, 113–114. https://ieeexplore.ieee.org/document/5779071/

K. Tumer and D.H. Wolpert. 2004. *Collectives and the Design of Complex Systems.* Springer.

Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111. https://doi.org/10.1145/79173.79181

Robbert van Renesse. 1998. Goal-oriented programming, or composition using events, or threads considered harmful. In *Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications 1998.* ACM, 82–87. https://doi.org/10.1145/319195.319208

Peter Van Roy. 2009. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music* 104 (2009), 616–621.

Joshua Cherian Varughese, Hannes Hornischer, Payam Zahadat, Ronald Thenius, Franz Wotawa, and Thomas Schmickl. 2020. A swarm design paradigm unifying swarm behaviors using minimalistic communication. *Bioinspiration & Biomimetics* 15, 3 (2020), 036005. https://doi.org/10.1088/1748-3190/ab6ed9

Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. 2019. From distributed coordination to field calculus and aggregate computing. *J. Log. Algebraic Methods Program.* 109 (2019). https://doi.org/10.1016/j.jlamp.2019.100486

Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, and Mats Helander et al. 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org.

Hiroshi Wada, Pruet Boonma, and Junichi Suzuki. 2007. A SpaceTime Oriented Macroprogramming Paradigm for Push-Pull Hybrid Sensor Networking. In *Proceedings of the 16th International Conference on Computer Communications and Networks, IEEE ICCCN 2007.* IEEE, 868–875. https://doi.org/10.1109/ICCCN.2007.4317927

Hiroshi Wada, Pruet Boonma, and Junichi Suzuki. 2008. Macroprogramming Spatio-temporal Event Detection and Data Collection in Wireless Sensor Networks: An Implementation and Evaluation Study. In *41st Hawaii International Conference on Systems Science (HICSS-41'08), Proceedings.* IEEE Computer Society, 498. https://doi.org/10.1109/HICSS.2008.237

Hiroshi Wada, Pruet Boonmab, and Junichi Suzukic. 2010. Chronus: A spatiotemporal macroprogramming language for autonomic wireless sensor networks. In *Autonomic Network Management Principles: From Concepts to Applications.* Academic Press, 167.

Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. 2015. Partitioned Global Address Space Languages. *ACM Comput. Surv.* 47, 4 (2015), 62:1–62:27. https://doi.org/10.1145/2716320

Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. 1996. A Note on Distributed Computing. In *Mobile Object Systems - Towards the Programmable Internet, 2nd International Workshop, MOS'96, Selected Presentations and Invited Papers (Lecture Notes in Computer Science),* Vol. 1222. Springer, 49–64. https://doi.org/10.1007/3-540-62852-5_6

Fei-Yue Wang, Kathleen M. Carley, Daniel Zeng, and Wenji Mao. 2007. Social Computing: From Social Informatics to Social Intelligence. *IEEE Intell. Syst.* 22, 2 (2007), 79–83. https://doi.org/10.1109/MIS.2007.41

Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. 2020. A Survey of Multitier Programming. *ACM Comput. Surv.* 53, 4 (2020), 81:1–81:35. https://doi.org/10.1145/3397495

Matt Welsh and Geoffrey Mainland. 2004. Programming Sensor Networks Using Abstract Regions. In *1st Symposium on Networked Systems Design and Implementation (NSDI'04), Proceedings.* USENIX, 29–42. https://dl.acm.org/doi/10.5555/1251175.1251178

Danny Weyns, M. Usman Iftikhar, Didac Gil de la Iglesia, and Tanvir Ahmad. 2012. A survey of formal methods in self-adaptive systems. In *5th International C* Conference on Computer Science & Software Engineering, C3S2E 2012.* ACM, 67–79. https://doi.org/10.1145/2347583.2347592

Kamin Whitehouse, Cory Sharp, David E. Culler, and Eric A. Brewer. 2004. Hood: A Neighborhood Abstraction for Sensor Networks. In *2nd International Conference on Mobile Systems, Applications, and Services (MobiSys'04), Proceedings.* ACM / USENIX. https://doi.org/10.1145/990064.990079

Kamin Whitehouse, Feng Zhao, and Jie Liu. 2006. Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data. In *Wireless Sensor Networks, 3rd European Workshop (EWSN'06), Proceedings (LNCS),* Vol. 3868. Springer, 5–20. https://doi.org/10.1007/11669463_4

Tom De Wolf and Tom Holvoet. 2004. Emergence Versus Self-Organisation: Different Concepts but Promising When Combined. In *Engineering Self-Organising Systems, Methodologies and Applications (LNCS),* Vol. 3464. Springer, 1–15. https://doi.org/10.1007/11494676_1

Michael J. Wooldridge. 2009. *An Introduction to MultiAgent Systems, Second Edition.* Wiley.

Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, and Fatemeh Jalali et al. 2019. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *J. Syst. Archit.* 98 (2019), 289–330. https://doi.org/10.1016/j.sysarc.2019.02.009

Viktor Zykov, Efstathios Mytilinaios, Mark Desnoyer, and Hod Lipson. 2007. Evolved and Designed Self-Reproducing Modular Robotics. *IEEE Trans. Robotics* 23, 2 (2007), 308–319. https://doi.org/10.1109/TRO.2007.894685