Alma Mater Studiorum · Università di Bologna

DISI – Dipartimento di Informatica: Scienza e Ingegneria

Dottorato di Ricerca in
Computer Science and Engineering

Ciclo XXXII

Settore Scientifico Disciplinare: ING-INF/05
Settore Concorsuale: 09/H1

Engineering Self-Adaptive Collective Processes for Cyber-Physical Ecosystems

Candidato:
Dott. Roberto Casadei

Supervisore:
Chiar.mo Prof. Ing. MIRKO VIROLI

Coordinatore Dottorato:
Chiar.mo Prof. DAVIDE SANGIORGI

Esame finale anno 2020

Contents

List of Figures

List of Tables

Acknowledgements

To my family, for their unconditional support
To my brothers Stefano and Marco, for their example
To my beloved fiancée Valentina, for being my side in this and other journeys

I wish to express my sincere gratitude to all the great researchers that have shared with me their time and thoughts. I especially need to mention the components of the software engineering research group in Cesena, that, since my Bachelor's studies, had a great influence on my formation as an engineer and scientist. These, in particular, include Proff. Antonio Natali, Andrea Omicini, Alessandro Ricci, and Mirko Viroli. They have been truly enlightening and inspirational, and I am grateful to them. In particular, Antonio – the ultimate *maestro* – made me understand engineering and the very fundamental methodological principles of computer science; his attitude and words have been illuminating and game changer. Andrea showed me the art of systematic conceptual development; moreover, his seminal, spontaneous references to the behind-the-scenes of University shed the light on the potential directions that a motivated, ambitious graduate student like I was could pursue. Alessandro and Mirko, in addition to having been my BEng and MEng thesis supervisors, respectively, showed me different stances to research. I must doubly thank Mirko, for his precious guidance and teachings (at every level) in my PhD activity. A special thanks also goes to my colleague and friend Danilo Pianini, for our tight teamwork and interactions. I also cannot refrain from thanking Giorgio Audrito, Ferruccio Damiani, Jacob Beal, Alessandro Aldini, Claudio Savaglio, Giancarlo Fortino, Simon Dobson, Schahram Dustdar, Christos Tsigkanos, Stefano Mariani, Giovanni Ciatto, Andrea Roli, Michele Braccini, Antonio Magnani—every one of them gave me something (teachings, discussions, chats, examples, ideas) along this path.

Abstract (italiano)

Con lo sviluppo di informatica e intelligenza artificiale, la diffusione pervasiva di *device* computazionali e la crescente interconnessione tra elementi fisici e digitali, emergono innumerevoli opportunità per la costruzione di sistemi socio-tecnici di nuova generazione. Tuttavia, l'ingegneria di tali sistemi presenta notevoli sfide, data la loro complessità—si pensi ai livelli, scale, eterogeneità, e interdipendenze coinvolti. Oltre a dispositivi *smart* individuali, collettivi cyber-fisici possono fornire servizi o risolvere problemi complessi con un "effetto sistema" che emerge dalla coordinazione e l'adattamento di componenti fra loro, l'ambiente e il contesto. Comprendere e costruire sistemi in grado di esibire *intelligenza collettiva* e *capacità autonomiche* è un importante problema di ricerca studiato, ad esempio, nel campo dei *sistemi collettivi adattativi*. Perciò, traendo ispirazione e partendo dall'attività di ricerca su coordinazione, sistemi multiagente e *self-**, modelli di computazione spazio-temporali e, specialmente, sul recente paradigma di *programmazione aggregata*, questa tesi tratta concetti, metodi, e strumenti per l'ingegneria di *ensemble* di elementi situati eterogenei che devono essere in grado di lavorare, adattarsi, e auto-organizzarsi in modo decentralizzato. Il contributo di questa tesi consiste in quattro parti principali. In primo luogo, viene definito e implementato un linguaggio di programmazione aggregata (ScaFi), interno al linguaggio Scala, per descrivere comportamenti collettivi e adattativi secondo l'approccio dei *campi computazionali*. In secondo luogo, si propone e caratterizza l'astrazione di *processo aggregato* per rappresentare computazioni collettive dinamiche concorrenti, formalizzata come estensione al *field calculus* e implementata in ScaFi. Inoltre, si analizza e implementa un prototipo di *middleware* per sistemi aggregati, in grado di supportare più stili architetturali. Infine, si applicano e valutano tecniche di programmazione aggregata in scenari di *edge computing*, e si propone un pattern, *Self-Organising Coordination Regions*, per supportare, in modo decentralizzato, attività decisionali e di regolazione in ambienti dinamici.

Parole chiave — *intelligenza collettiva computazionale ; processi collettivi ; sistemi multiagente; sistemi cyber-fisici; auto-adattatività, auto-organizzazione ; coordinazione.*

Abstract

The pervasiveness of computing and networking is creating significant opportunities for building valuable socio-technical systems. However, the scale, density, heterogeneity, interdependence, and QoS constraints of many target systems pose severe operational and engineering challenges. Beyond individual smart devices, cyber-physical collectives can provide services or solve complex problems by leveraging a "system effect" while coordinating and adapting to context or environment change. Understanding and building systems exhibiting *collective intelligence* and *autonomic capabilities* represent a prominent research goal, partly covered, e.g., by the field of *collective adaptive systems*. Therefore, drawing inspiration from and building on the long-time research activity on coordination, multi-agent systems, autonomic/self-* systems, spatial computing, and especially on the recent aggregate computing paradigm, this thesis investigates concepts, methods, and tools for the engineering of possibly large-scale, heterogeneous *ensembles* of situated components that should be able to operate, adapt and self-organise in a decentralised fashion. The primary contribution of this thesis consists of four main parts. First, we define and implement an aggregate programming language (ScaFi), internal to the mainstream Scala programming language, for describing collective adaptive behaviour, based on field calculi. Second, we conceive of a "dynamic collective computation" abstraction, also called *aggregate process*, formalised by an extension to the field calculus, and implemented in ScaFi. Third, we characterise and provide a proof-of-concept implementation of a middleware for aggregate computing that enables the development of aggregate systems according to multiple architectural styles. Fourth, we apply and evaluate aggregate computing techniques to edge computing scenarios, and characterise a design pattern, called *Self-organising Coordination Regions (SCR)*, that supports adjustable, decentralised decision-making and activity in dynamic environments.

Keywords — *computational collective intelligence ; collective processes ; multi-agent systems ; cyber-physical systems ; self-adaptive, self-organising systems ; coordination.*

Chapter 1

Introduction

Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura
ché la diritta via era smarrita

Dante Alighieri · *Divina Commedia*, Inferno, Canto I

Contents

1.1 Research Context and Motivation

In these years, we are witnessing huge technological advances in ICT (Information and Communication Technology), creating disruptions at various levels: the pervasiveness of computers and connectivity, the establishment of the paradigm of the cloud/fog/edge computing continuum, the software-defined everything, the

culture of devops, the explosion and popularisation of artificial intelligence and machine learning, the prospective of 5g, etc. Various buzzwords are used to denote different trends of these fervent times.

internet of things

The *internet of things (IoT)* promises to *bridge* the *physical world* (of everyday objects) with the *digital world* (of computers), hence extending the connectivity that Internet granted to computers up to just everything (from ourselves to the entities of our environments). Similarly, the field of *cyber-physical systems* is devoted to the study of distributed systems involving connected physical and computational processes (often regulated through feedback loops).

cyber-physical systems

big data

As potentially everything can be seen as a producer or consumer (or both—i.e., *prosumer*) of information, the problem of *big data* – i.e., the volume, velocity, variety, veracity of data – demands for efficiency in the use of computational power. Therefore, it becomes important to consider *where* computation is needed and provided, to avoid large bandwidth usage and latency. Accordingly, the *fog* and *edge computing* paradigms aim at providing cloud computing-like functionality "at the edge of the network", i.e., in close proximity to where data is generated or resources are needed, hence supporting real-time computation (by reducing the latency of communication with respect to remote data centres) and providing computation and storage where cloud access is not possible.

fog/edge computing

The inexorable growth in scale, density, smart-ness, and interrelation of artificial systems is challenging the human ability of anticipating situations, understanding (the reasons behind) phenomena, and carrying out timely corrective interventions. This prevision of an exploding complexity is not new. In 2003, Kephart and Chess already noted that *"soon systems will become too massive and complex for even the most skilled system integrators to install, configure, optimize, maintain, and merge"* and accordingly proposed *autonomic computing* [KC03] as a solution, i.e., a paradigm where *"computing systems can manage themselves given high-level objectives from administrators"*. Endowing self-adaptivity and self-organisation capabilities to artificial systems is challenging, but inspiration can be taken by looking at how natural systems implement them.

autonomic computing

Beyond the realm of technology progress, the general goal of engineering is the sustainable development of efficient *socio-technical systems* – i.e., systems made of humans and machines operating in some environment – able to effectively provide

socio-technical systems

value. Effectively integrating humans and other physical or artificial entities into synergetic, collaborative systems is a long-standing (and exciting) issue in engineering. *Collective computing*, identified by Abowd [Abo16] as the fourth generation in computing (after Weiser's characterisation of the computing evolution from mainframe to personal and ubiquitous computing [Wei91]), represents a paradigm where heterogeneous collectives of humans and artificial components synergically cooperate to solve complex problems.

collective computing

1.2 Overview and Contribution

The technological context outlined in Section 1.1 is a source of complexity and challenges for which traditional engineering approaches seem to fall short. Our world is made of (possibly very-large scale) networks of (possibly heterogeneous) artificial and natural elements, situated in one or more environments, which can be leveraged in order to provide value; to do so, they need to coordinate, self-organise, self-adapt to change, and so on.

Therefore, this work is mostly concerned with *computational collective intelligence*, i.e., *"the form of intelligence that emerges from the collaboration and competition of many individuals (artificial and/or natural)"* [NKC09], or, similarly, with the field of *collective adaptive systems* [And+13]. This work draws inspiration from and builds on the long-time research activity on coordination [MC94], multi-agent systems [Woo09], autonomic/self-* systems [KC03], spatial computing [Bea+13], collective adaptive systems [And+13], and especially on the recent aggregate computing paradigm [BPV15].

computational collective intelligence

1.2.1 General problem statement

> What concepts, methods, and tools can help in the engineering of computational collective intelligence? The problem comprises the analysis, design, implementation, evaluation, and deployment of possibly large-scale, heterogeneous collective adaptive systems, i.e., large ensembles of situated components that should be able to operate, adapt and self-organise in a decentralised fashion.

1.2.2 Specific problem statement

The state of the art in aggregate computing research and related fields provides perspectives and challenges related to modelling, development, and operation of collective cyber-physical systems. Therefore, how to model and design heterogeneous situated collectives? How to compositionally and declaratively specify their self-adaptive, self-organising behaviour (in particular, providing functionality while dynamically exploiting opportunities arising, with "loose assumptions" on the autonomy, reliability and connectivity of components)? Finally, how to develop and operate such systems considering modern programming environments and the emerging multi-layer architectures?

1.2.3 Contributions

This thesis illustrates four primary contributions:

1) definition and implementation of a language (ScaFi), embedded into the mainstream Scala programming language, for describing collective adaptive behaviour, based on field calculi;
2) conception of a "dynamic collective computation" abstraction, also called *aggregate process*, formalised as an extension to the field calculus, and implemented in ScaFi;
3) design and proof-of-concept implementation of a middleware for aggregate computing, allowing development of aggregate systems according to multiple architectural styles.
4) application and evaluation of aggregate computing techniques to edge computing scenarios, and proposal of a design pattern, called *Self-organising Coordination Regions (SCR)*, to support adjustable, decentralised decision-making and activity in dynamic environments.

Moreover, the reader can also find, as secondary contribution, an up-to-date literature review on collective adaptive systems and related fields from the perspective of software engineering.

References

[Abo16] Gregory D Abowd. "Beyond weiser: From ubiquitous to collective computing". In: *Computer* 49.1 (2016), pp. 17–23.

[And+13] S Anderson, N Bredeche, AE Eiben, G Kampis, and MR van Steen. "Adaptive collective systems: herding black sheep". In: (2013).

[Bea+13] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. "Organizing the Aggregate: Languages for Spatial Computing". In: *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. A longer version available at: `http://arxiv.org/abs/1202.5509`. IGI Global, 2013. Chap. 16, pp. 436–501. ISBN: 978-1-4666-2092-6. DOI: `10.4018/978-1-4666-2092-6.ch016`.

[BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. "Aggregate Programming for the Internet of Things". In: *IEEE Computer* 48.9 (2015), pp. 22–30. DOI: `10.1109/MC.2015.261`.

[KC03] Jeffrey O Kephart and David M Chess. "The vision of autonomic computing". In: *Computer* 1 (2003), pp. 41–50.

[MC94] Thomas W Malone and Kevin Crowston. "The interdisciplinary study of coordination". In: *ACM Computing Surveys (CSUR)* 26.1 (1994), pp. 87–119.

[NKC09] Ngoc Thanh Nguyen, Ryszard Kowalczyk, and Shyi-Ming Chen. "Computational Collective Intelligence. Semantic Web, Social Networks and Multiagent Systems". In: *Conference proceedings ICCCI*. Springer. 2009, p. 269.

[Wei91] Mark Weiser. "The Computer for the 21 st Century". In: *Scientific american* 265.3 (1991), pp. 94–105.

[Woo09] Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

1.3 About This Thesis

This thesis is organised as follows.

Chapter 1 provides a brief introduction about the motivation, scope, structure, and results of this work.

Part I provides some background in terms of perspectives, concepts, and the state of the art. This background and the main conceptual elements are first outlined in **Chapter 2**. Then, more detail is provided in the areas of distributed co-

ordination (**Chapter 3**), spatial and collective adaptive computing (**Chapter 4**), aggregate computing (**Chapter 5**) – which is the core research thread of this work –, and infrastructures and deployments for modern system (**Chapter 6**).

Part II provides the contribution of this thesis, which consists of four main parts. **Chapter 7** covers ScaFi, an aggregate programming language and toolkit, internal to the Scala programming language. **Chapter 8** covers *aggregate processes*, an aggregate computing abstraction for describing dynamic collective computations on dynamic domains of devices. **Chapter 9** discusses *aggregate computing platforms*, and presents a middleware for building aggregate systems. **Chapter 10** shows application of aggregate programming techniques to edge computing scenarios and presents a design pattern for large-scale, dynamic ecosystems.

Finally, **Chapter 11** provides a final discussion, draws conclusions, and presents perspectives for further work.

1.4 List of Publications

2016

1. Roberto Casadei and Mirko Viroli. "Towards Aggregate Programming in Scala". In: *First Workshop on Programming Models and Languages for Distributed Computing*. ACM. 2016, p. 5
2. Roberto Casadei, Danilo Pianini, and Mirko Viroli. "Simulating large-scale aggregate MASs with alchemist and scala". In: *Computer Science and Information Systems (FedCSIS), 2016 Federated Conference on*. IEEE. 2016, pp. 1495–1504
3. Mirko Viroli, Roberto Casadei, and Danilo Pianini. "On execution platforms for large-scale aggregate computing". In: *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. ACM. 2016, pp. 1321–1326
4. Giorgio Audrito, Ferruccio Damiani, Mirko Viroli, and Roberto Casadei. "Run-Time Management of Computation Domains in Field Calculus". In: *Foundations and Applications of Self* Systems, IEEE International Workshops on*. IEEE. 2016, pp. 192–197

2017

1. Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. "Compositional Blocks for Optimal Self-Healing Gradients". In: *Self-Adaptive and Self-Organising Systems (SASO), IEEE International Conference on*. IEEE. 2017
2. Roberto Casadei, Alessandro Aldini, and Mirko Viroli. "Combining Trust and Aggregate Computing". In: *Foundations of Coordination Languages and Self-Adaptative Systems (FOCLASA), IEEE International Workshop on*. IEEE. 2017

2018

1. Roberto Casadei, Giancarlo Fortino, Danilo Pianini, Wilma Russo, Claudio Savaglio, et al. "Modelling and Simulation of Opportunistic IoT Services with Aggregate Computing". In: *Future Generation Computer Systems* 91 (2018), pp. 252–262. ISSN: 0167-739X. DOI: 10.1016/j.future.2018.09.005
2. Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, et al. "From Field-Based Coordination to Aggregate Computing". In: *Proceedings of the 2018 International Conference on Coordination Models and Languages*. 2018, pp. 252–279
3. Roberto Casadei and Mirko Viroli. "Programming Actor-Based Collective Adaptive Systems". In: *Programming with Actors: State-of-the-Art and Research Perspectives*. Vol. 10789. Lecture Notes in Computer Science. Springer, 2018, pp. 94–122. DOI: 10.1007/978-3-030-00302-9_4
4. Roberto Casadei, Alessandro Aldini, and Mirko Viroli. "Towards Attack-Resistant Aggregate Computing Using Trust Mechanisms". In: *Science of Computer Programming* 167 (2018), pp. 114–137. DOI: 10.1016/j.scico.2018.07.006
5. Danilo Pianini, Giovanni Ciatto, Roberto Casadei, Stefano Mariani, Mirko Viroli, et al. "Transparent Protection of Aggregate Computations from Byzantine Behaviours via Blockchain". In: *Proceedings of the 4th EAI International Conference on Smart Objects and Technologies for Social Good*.

ACM. 2018, pp. 271–276

6. Roberto Casadei and Mirko Viroli. "Collective Abstractions and Platforms for Large-Scale Self-Adaptive IoT". in: *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE. 2018, pp. 106–111

2019

1. Roberto Casadei, Giancarlo Fortino, Danilo Pianini, Wilma Russo, Claudio Savaglio, et al. "A development approach for collective opportunistic Edge-of-Things services". In: *Information Sciences* 498 (2019), pp. 154–169
2. Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, et al. "From distributed coordination to field calculus and aggregate computing". In: *Journal of Logical and Algebraic Methods in Programming* (2019), p. 100486. ISSN: 2352-2208. DOI: `https://doi.org/10.1016/j.jlamp.2019.100486`
3. Roberto Casadei, Mirko Viroli, Giorgio Audrito, Danilo Pianini, and Ferruccio Damiani. "Aggregate Processes in Field Calculus". In: *Coordination Models and Languages*. Ed. by Hanne Riis Nielson and Emilio Tuosto. Cham: Springer International Publishing, 2019, pp. 200–217. ISBN: 978-3-030-22397-7
4. Roberto Casadei, Danilo Pianini, Mirko Viroli, and Antonio Natali. "Self-organising Coordination Regions: A Pattern for Edge Computing". In: *Coordination Models and Languages*. Ed. by Hanne Riis Nielson and Emilio Tuosto. Cham: Springer International Publishing, 2019, pp. 182–199. ISBN: 978-3-030-22397-7
5. Roberto Casadei, Christos Tsigkanos, Mirko Viroli, and Schahram Dustdar. "Engineering Resilient Collaborative Edge-Enabled IoT". in: *2019 IEEE International Conference on Services Computing (SCC)*. 2019, pp. 36–45. DOI: `10.1109/SCC.2019.00019`
6. Roberto Casadei and Mirko Viroli. "Coordinating Computation at the Edge: a Decentralized, Self-Organizing, Spatial Approach". In: *2019 Fourth Inter-*

national Conference on Fog and Mobile Edge Computing (FMEC). 2019, pp. 60–67. DOI: 10.1109/FMEC.2019.8795355

7. Danilo Pianini, Roberto Casadei, and Mirko Viroli. "Security in Collective Adaptive Systems: A Roadmap". In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE. 2019, pp. 86–91
8. Roberto Casadei, Danilo Pianini, Guido Salvaneschi, and Mirko Viroli. "On Context-Orientation in Aggregate Programming". In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE. 2019, pp. 92–97

Submitted

1. Roberto Casadei, Mirko Viroli, Giorgio Audrito, and Ferruccio Damiani. "Aggregate Programming in Scala with ScaFi". In: 2020. Submitted to a journal.

Part I

Background and Motivation

Chapter 2

Towards Collective Adaptive Computing: Concepts and Perspectives

In the case of all things which have several parts and in which the totality is not, as it were, a mere heap, but the whole is something beside the parts, there is a cause.

Aristotle · *Metaphysics*

Contents

In [Abo16], Abowd identifies *collective computing* as the fourth generation in computing, after mainframe, personal, and ubiquitous computing as identified by Weiser in his seminal paper [Wei91]. In this historical and conceptual characterisation, computing generations are distinguished by the human-computer interaction patterns they enable, the "canonical devices" involved, and the corresponding opportunities in terms of applications:

- *1st generation* (1930s) — many operators interact with a single mainframe, e.g., for scientific computing and data processing;
- *2nd generation* (1960s) — each user has one personal computer for carrying out a wide variety of tasks;
- *3rd generation* (1980s) — each user is provided with contextual services by many computers distributed in the environment (space becomes the device);
- *4th generation* (2000s) — many-to-many human-computer ratio fostered by the synergy of (i) cloud computing, (ii) crowd computing, and (iii) the IoT, bridging the digital and the physical worlds through networks.

The problem of capturing, computationally, the behaviour of *collectives* of entities is central to several research threads. At the core, the problem relates the *local* with the *global*, the *micro* with the *macro*, *individual activity* with *collective activity*, and to the problem of *collective intelligence*. To the matter of its characterisation, interaction (and its ruling, i.e., coordination) is obviously crucial, since it represents the basic mechanism by which *parts* can communicate with and affect other parts and ultimately the *whole*. For instance, in Multi-Agent Systems (MAS) research, agents are often defined as *social* entities [Cas98], to stress the importance of interaction both at the individual and the system level and foster the existence of explicit structures (e.g., enforced through norms or laws) to induce order and interesting properties in the system. In this chapter, various fields and notions related to systems, collective and adaptive computing are reviewed. An overview of the relationships between the surveyed perspectives is provided by Figure 2.1.

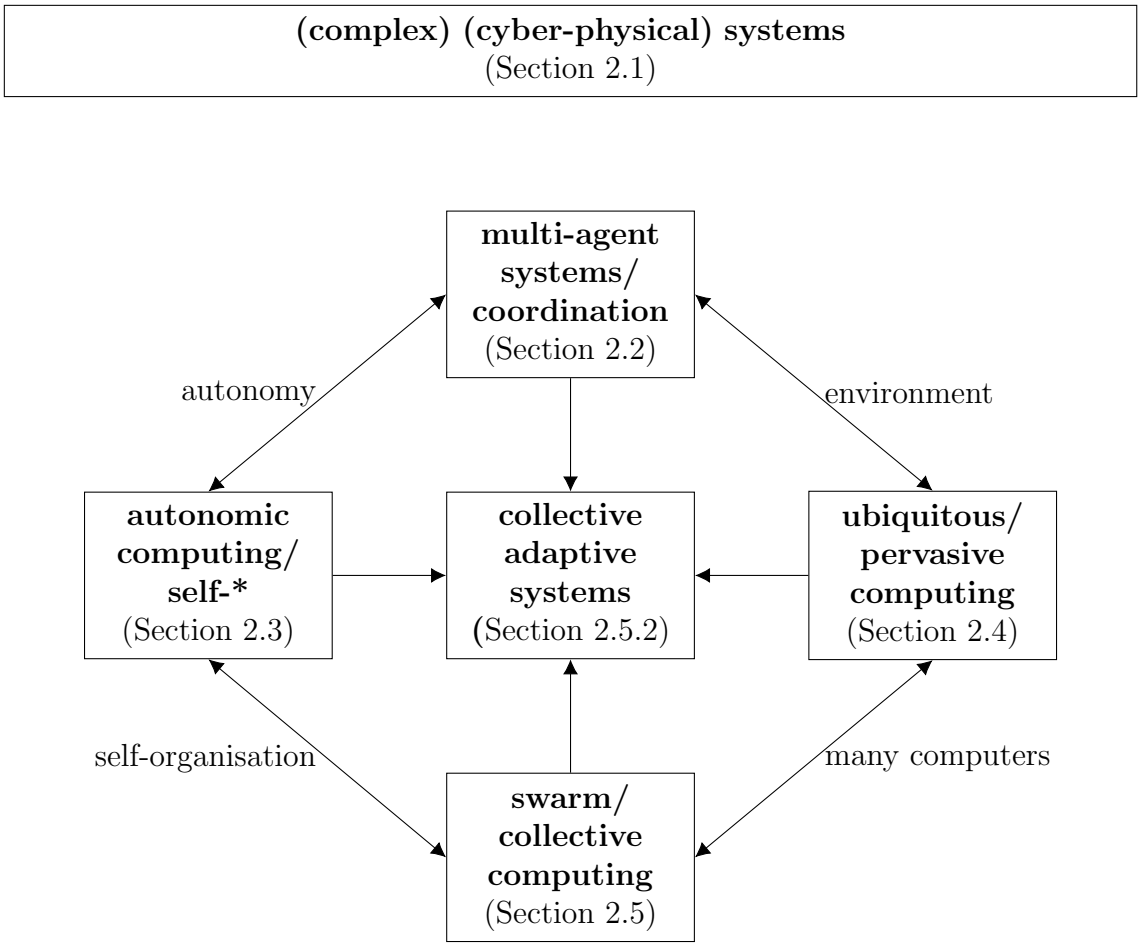


Figure 2.1: Collective adaptive systems research and related fields.

2.1 (Complex) Systems

The notion of a *system*, i.e., an organised whole made of a set of elements that interact in a non-trivial way, is now mainstream in science and engineering. Even though such a notion can be traced back in the ancient past, the paradigm of *systems thinking* is a relatively recent achievement in human history, fostered by a set of theories and research threads that started around the mid of the 20th century, e.g., especially, Bertalanffy's *general systems theory* [Ber69], Weiner's *cybernetics* [Wie65; Ash61], artificial intelligence, dynamical systems theory, and *complexity science* [Wea48].

system thinking

The abstract notion of system and its related concepts provide important cognitive tools for scientists and engineers. In the *Systems Praxis Framework* [Sin+12], *system thinking* bridges the foundations, theories, and representations of the *integrative systems science* with the *hard/soft systems approaches to practice* [Sil12; Che00]. *Systems science* [MK15] (also known as *systemics* or *systems theory*) is the integrative, interdisciplinary field that studies systems and that fosters a perspective through which the world can be seen as a *system of systems (SoS)* [Mai98] or a set of *networked, interwoven systems* [Tom+14].

systems science

A system may be defined, as e.g. in [MK15], as

> *a whole made up of interacting or interdependent elements or components integrally related among themselves in a way that differs from the relationships they may have with other elements.*

system

So, as per [MK15], a system can be conceptualised as a *bounded object*, with concrete or conceptual, fuzzy or sharp, variously porous *boundaries* (e.g., defined as per some *boundary condition*), that is *organised* (with a *structure*, or persistent connectivity pattern among components, promoting some *function* or *process*) and involved in the exchange of *flows* of matter, energy, and information with its *environment*. The notion of environment is especially important in the related notions of *ecology* (from Greek *oikos*, "house" or "environment", and *logos*, "study of") and *ecological system* (or *ecosystem* for short) [CIMV11], i.e., a system of components that interact with one another and with their surrounding environment—which consists of living, or biotic (i.e., active), and non-living, or abiotic (i.e., passive) components.

ecosystem

A particular class of systems that has been recently given much attention is that of so-called *complex systems*, i.e., systems that exhibit *complexity*. Etymologically, *complex* means "intertwined", hence pertaining to a situation which is harder to solve than another one which is *simple* (i.e., "without any folds") or *complicated* (i.e., "with folds"); in other words, whereas simple problems are onefold and complicated problems can be unfolded, complex problems are metaphorically knots and need to be untangled in a non-obvious way. According to [BY02],

complex systems

complex vs. complicated

> *"Complex Systems" is the new approach to science studying how relationships between parts give rise to the collective behaviors of a system, and how the system interacts and forms relationships with its environment.*

Complexity shows the limitations of *reductionistic approaches*, since complex systems have *emergent properties* which *"cannot be understood or predicted simply by analysing the structure of their components"* [VR04]. This is coherent to the old saying that "a whole is more than the sum of its parts" (Aristotle)—which also suggests that complexity *emerges* from non-linear interactions (cf., *nonlinear science* [NN95]). In contrast to reductionism, *holism* (from Greek "holos", which means "whole") is the paradigm that focusses on the whole and its internal and external relationships. Key properties of complex systems such as adaptation and emergence are discussed in the rest of this chapter.

reductionism

holism

By definition, understanding, predicting, and controlling complex systems is not easy, but what about engineering complex systems? The nature of this challenge is effectively described by Ottino in [Ott04].

> *The hallmarks of complex systems are adaptation, self-organization and emergence [...] And this is where the conceptual conflict with engineering arises. Engineering is not about letting systems be. Engineering is about making things happen, about convergence, optimum design and consistency of operation [...] Engineering is about assembling pieces that work in specific ways— that is, designing complicated systems.*

Still, something can be done.

> *Engineers calculate, and calculation requires a theory, or at least an organized framework. Could there be laws governing complex systems? If*

> *by 'laws' one means something from which consequences can be derived – as in physics – then the answer may be no. But how about a notch below, such as discovering relationships with caveats, as in the ideal gas 'law', or uncovering power-law relationships? Then the answer is clearly yes.*

2.1.1 Cyber-physical systems

With the pervasiveness of computing (see Section 2.4), systems consisting of both digital (or *cyber*) and physical components tend to emerge: these are called *cyber-physical systems (CPS)*.

cyber-physical systems

Gill defines CPSs [Gil08] as

> *physical, biological, and engineered systems whose operations are integrated, monitored, and/or controlled by a computational core. Components are networked at every scale. Computing is deeply embedded into every physical component, possibly even into materials. The computational core in an embedded system, usually demands real-time response, and is most often distributed.*

Indeed, CPSs consist of three main kinds of components [Gun+14]:

1. *physical elements* — monitored and/or controlled;
2. *interfaces* — networking components and other intermediaries most notably including sensors, actuators, analog-to-digital (ADC) and digital-to-analog (DAC) converters;
3. *cyber, embedded devices* — processing information and interacting among them and with their environment;

and are often characterised by distribution, real-time operation, and feedback loops. Peculiar aspects such as, for instance, irreversibility and non-preemptability of actuations, must be taken into account and are in many cases sources of challenges.

2.2 Multi-Agent Systems

autonomous agents

Agents are *autonomous* computational entities [FG96]. The agent abstraction is a powerful one for dealing with the increasing complexity of current and future software-based systems. It helps to fill the gap created by the very peculiar trends of this historical period—namely the growing interconnection, pervasiveness of computing, delegation to machines, human-orientation and intelligence endowed to artificial systems [Woo09].

Different research fields study or consider the notion of an agent, adopting a specific viewpoint while retaining some common concepts (fundamentally, autonomy and consequential features). In Artificial Intelligence (AI), the focus is on the development of cognitive intelligent agents, with the problem of representing the world in a symbolic way and carrying on "intelligent processes" (e.g., reasoning, planning). In Distributed AI (DAI) [FW99], the agent is seen as a (distributed) component of a *Multi-Agent System (MAS)*, *situated* into an *environment* and interacting with the environment as well as with other agents. The aspect of *situatedness* is, together with mobility and spatial reasoning/operation, also crucial in robotics. By a programming language (PL) perspective, since agents encapsulate invocation (i.e., they are autonomous), they can be thought of as the next step of an evolution from monoliths to modules (encapsulation of behaviour), to objects (encapsulation of state in addition to behaviour) [Par97; Ode02], to active objects/actors (decoupling invocation from execution). By the point of view of software engineering, agents and related abstractions are considered as useful tools for the analysis and design of software systems—see also *agent-oriented software engineering (AOSE)* [Woo97].

multi-agent system

situatedness

AOSE

autonomy

The key distinguishing property of agents is *autonomy*. Autonomy is a complex, multifaceted notion which essentially refers to the ability of an agent to govern itself (e.g., its own activities, goals, and other aspects, depending on the notion and degree of autonomy considered). Autonomy, together with *agency* (i.e., the ability to act), requires some form of *proactivity* (i.e., the ability to take the initiative) and context-awareness, as well as the ability of (inter)acting in some environment and society. These characteristics are sufficient for a *weak* notion of agent; *strong* agents, in addition, are intelligent in the sense that they explicitly

agency

proactivity

weak vs. strong agents

represent their goals, have mental states, and exhibit cognitive capabilities. The *Belief-Desire-Intention (BDI) model* and *architecture* [RG95] provide a practical support for a strong notion of agency.

There are two key problems in the use and development of agents: the design of *individual* agents (*micro level*) and the design of a *society* of agents (*macro level*). The focus on the macro level is natural since generally the goal of a MAS is not just putting agents together but possibly increasing individual performance (optimisation by collaboration) and enabling agents to accomplish tasks that go beyond individual skills (extending capabilities).

micro vs. macro

environment

In addition, the notion of *environment* is considered a fundamental abstraction for MASs [WOO07; Vir+07]. According to [Pla+07], the two main functions of the environment abstraction are *coordination* (e.g., through mechanisms for decoupled communication, synchronisation, and overlay structures) and *resource and context management*. The *Agents&Artefacts (A&A) meta-model* [ORV08] fosters a first-class approach to society and environment design through the notion of *artefact*, i.e., passive or reactive entities that mediate agent-to-agent and agent-environment interactions. Artefacts can support forms of coordination based on *cognitive stigmergy* and self-organisation (cf., co-fields—see Chapter 3).

coordination

When it comes to coordination in MASs, multiple approaches can be considered, such as subjective coordination [OO03] via agent-to-agent communication, objective coordination [OO03] through coordination artefacts [Omi+04], organisation/normative models, and self-organisation.

2.2.1 Main aspects in MASs

Coordination As anticipated, the research line of MAS inherently acknowledges the key role of coordination [NLJ96a] by focussing on the macro level of systems of interacting autonomous agents. One key coordination challenge is to make agents *cooperate* despite conflicting goals, e.g., through consistent multi-agent planning and proper negotiation. In [NLJ96b], coordination techniques are classified in four categories: *(i) organisational structuring*, *(ii) contracting* (see, e.g., Smith's *contract net protocol* [Smi80]), *(iii) multi-agent planning*, and *(iv) negotiation* (e.g., based on game theory). The survey in [Cao+13] provides an account of recent progress in distributed multi-agent coordination in the areas of consensus, forma-

tion control, optimisation, task assignment, and estimation.

Historically, coordination research begins with simple coordination of parallel activities, then moves towards increasing intelligence in coordination and distribution into increasingly complex self-organising distributed coordination systems. Chapter 3 provides a detailed account on this historical development.

Organisation Beside coordination, MAS research recognises the importance that the organisational dimension [HL04] assumes in the realisation of system-level behaviour. Indeed, the function of structure and order is to regulate interactions to achieve static or dynamic goals. In [HL04], Horling et al. survey the following major MAS organisational paradigms at the time: *(i) network organisations* or *adhocracies*, with complex and dynamic structures; *(ii) hierarchies*, with tree-like structures; *(iii) holarchies*, i.e., hierarchically nested structures of *holons* (which are both *wholes* and *parts*) with cross-tree interactions; *(iv) coalitions*, i.e., short-lived, goal-directed groups of agents with the goal of maximising individuals' utilities; *(v) teams*, i.e., sets of cooperative agents which have agreed to work together towards a common goal; *(vi) congregations*, i.e., long-lived agent groupings, formed with no specific goal in mind, aimed at facilitating the process of finding collaborators (cf., service discovery); *(vii) societies*, i.e., long-lived, open organisations aimed at providing consistency through social laws to facilitate coexistence and ordered-yet-flexible interaction; *(viii) federations*, i.e., groups of agents which have ceded some autonomy to a single delegate which represents the group and mediates interaction with other groups; *(ix) markets*, i.e., organisations of competitive buyers, suppliers, and sellers, mainly aimed at supporting processes of allocation and pricing; *(x) matrix organisations*, i.e., structures with rows of agents and columns of managers.

The significance of the organisational aspect has motivated the emergence of frameworks and linguistic approaches (grouped under the notion of *organisation-oriented programming* [BHS06]) to model the organisational dimension of MAS, such as e-institutions [Est+01] and $\mathcal{M}oise^+$ [HSB07].

organisation vs. institution

The concept of *institution*, e.g. in human organisations, is used to denote a set of formal and informal *social rules* (e.g., convention, habits, laws, etc.) for structuring and constraining agent interaction (i.e., for coordination). While

organisations can be defined as "social units (or human groupings) deliberately constructed and reconstructed to seek specific goals" [Etz64], institutions represent means for designing organisations. As a computational counterpart of institutions, *electronic institutions (e-institutions)* are institutionalised agent organisations [Est+01]. In the research line of *normative MASs* [HW11], approaches consider *normative agents* that are able to reason about, manipulate, and take decisions regarding *social norms* in order to foster social control and cooperation (e.g., through incentives or *sanctions* aimed at *minimising deviance*)—especially in open, heterogeneous systems [AP08].

electronic institutions

normative MAS

Coherently with the autonomic vision covered in Section 2.3, the perspective of self-organisation is particularly relevant in MAS [SGK05], as it provides a way to deal with change in the environment and system itself.

Other key aspects Other relevant aspects in MASs include the following:

- *Communication* — In order to *communicate*, agents need to agree upon the syntax and semantics of messages. *Ontologies*, as formal definitions of bodies of knowledge, are used for that. Then, agents may perform *communicative actions* to *influence* other agents.
- *Planning* — Agents must be able to decide what to do; i.e., they should possess *decision-making* capabilities directed towards activity (*practical reasoning*). Practical reasoning is a two-step process that consists of deciding what to achieve (*deliberation*) and then deciding how to achieve the prefigured *states of affairs* (*means-ends reasoning*).
- *Learning* — Choosing the "right" actions to perform can be difficult in uncertain environments. This could be tackled by *learning* and, in particular, by the approach known as *reinforcement learning*, which leverages trial-and-error, reward-oriented search to improve performance while balancing exploration and exploitation [RA18].

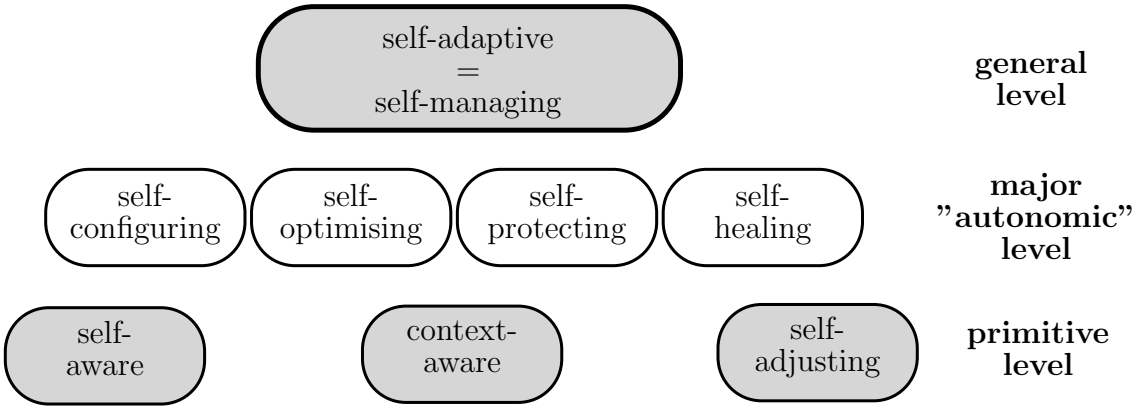


Figure 2.2: Hierarchy of self-* properties, adapted from [ST09].

2.3 Self-* Systems

2.3.1 Autonomic computing

autonomic computing

In the seminal paper on *autonomic computing* [KC03], Kephart and Chess foresee how the ability of computer-based systems to manage themselves according to human-dictated, high-level goals could be crucial to deal with the increasing complexity fuelled by recent trends (cf., massiveness, heterogeneity, dynamicity). Operationally, in a fast-changing world where quick (yet thoughtful and comprehensive) interventions are required, human-in-the-loop approaches tend to fall short, and the ideas of autonomous maintenance and upgrade become more and more appealing, fostering a vision of "eternal systems" [Mul12]. Regarding the engineering of self-adaptive software, it is important to define why/what/where/when change is required, who is responsible for implementing change, and how change can be implemented [ST09]. Architecturally, *autonomic elements* consist of *managed elements* controlled by an *autonomic manager* that implements *monitoring, analysis, planning, execution, and knowledge management* functions (MAPE-K) [KC03].

eternal systems

MAPE-K

2.3.2 Self-* properties

According to [KC03], the core of autonomic computing is *self-management*, which can be declined into *(i) self-configuration*, to cope with building and integration; *(ii) self-optimisation*, to improve operation; *(iii) self-healing*, to find,

diagnose and repair issues; and *(iv) self-protection*, to prevent and respond to internal and external attacks. Self-management is also called *self-adaptiveness*. In [Ore+99], self-adaptive software is defined as a software that

> *modifies its own behaviour in response to changes in its operating environment*

where operating environment denotes *"anything observable by the software system"*, which is also usually the definition of *context* as found, e.g., in the research fields of context-aware systems [AAC16] and context-oriented programming [HCN08]. Indeed, a system, in order to be able to adapt or manage itself, needs to be able to perceive (change in) internal and external states, and to change itself [HS06]. That is, a self-adaptive system must be *self-aware* and *context-aware* [ST09], in addition to *self-adjusting* [HS06]. Figure 2.2 provides a hierarchical view of self-* properties.

operating environment = context

Computationally, self-awareness and the ability to self-adjust can be achieved through *reflection mechanisms*. A reflective system can be defined as a system which *"can access and manipulate a full, explicit,* causally connected *representation of its own state"* [Mae87], where causal connection means that models and states are synchronised in both directions (from states to models through observation and from models to states through action). The combination of ideas from computational reflection and model-driven engineering has led to the research field of *models@runtime (MRT)* [BBF09; BGS19], which aims at supporting the development of long-lived, self-adaptive software through the use of *runtime models*.

reflection

models-at-runtime

In general, rigorous specification and validation of self-* systems can be supported through formal methods. According to survey [Wey+12], use of formal methods in this research area is increasing but still somewhat limited. Formalisms for modelling include algebra, (labelled) transition systems, process algebras, and formal specification languages. Concerning property specification and verification, the same modelling language may be used, or various kinds of logics (e.g., temporal or spatial logics).

Self-adaptive vs. adaptive vs. non-adaptive How can adaptivity be defined? How can one determine whether a system is adaptive or not? The verb

"to adapt" derives from Latin "adaptare" – which is made of "ad" (meaning "purposefully") and "aptare" (meaning "to adjust") – which means "to adjust one thing with respect to another thing according to convenience or proportion". So, adaptivity consists of an adaptation relationship between some subject and some goal or element of comparison. Thus, an "adaptive" system is one that "relates to adaptation" or "tends to adapt", i.e., that adjusts something (often itself, hence "self-adaptive") purposefully. However, it is important to distinguish "adaptivity" with "dynamics" or "activity". For instance, a thermostat that, while driven by the same rules, acts diversely based on the temperature of the moment, is not considered an adaptive system; i.e., an adaptive system is one that changes its control rules by experience [And+13]. Thus, a self-adaptive system can be defined as an adaptive system where the control rules are part of the system itself.

Emergence and self-organisation How does self-adaptiveness relate to the distinction between the micro and macro levels? Generally, self-adaptation is considered to be carried out as a top-down process. Self-organisation is often considered as the bottom-up counterpart of self-adaptation. In this work, however, self-organisation is intended as a structure-oriented flavour of self-adaptiveness and is distinguished from the bottom-up effect of emergence.

In [DWH04], *emergence* is defined as a property exhibited by a system *emergence*

> *when there are coherent emergents at the macro-level that dynamically arise from the interactions between the parts at the micro-level. Such emergents are novel w.r.t. the individual parts of the system.*

That is, emergence is characterised by novel, dynamic, robust, coherent patterns of micro-macro effects that raise when several parts interact in a decentralised way [DWH04]. The key point of emergence, which has consequences in the engineering of systems exhibiting it, is that it is hard to understand or track emergents back to the behaviour of the parts. For instance, sometimes, emergence is characterised by sensitivity to initial conditions or small parameter changes.

In [DWH04], *self-organisation* is defined as *self-organisation*

> *a dynamical and adaptive process where systems acquire and maintain structure themselves, without external control.*

Verb "to organise" derives from Latin "organum" (i.e., "organ") and hence its etymology is "to build organs", meaning *"to arrange several elements into a purposeful sequential or spatial (or both) order or structure"* (Business Dictionary). So, self-organisation is an autonomous, robust, flexible process that seeks an increase of order [DWH04].

As suggested in [DWH04], emergence and self-organisation are different concepts that may coexist or be fruitfully integrated by (i) making self-organisation emerge or (ii) supporting emergence of properties through self-organisation.

Emergence and engineering The ability to create qualitatively new, macro-level properties out of micro-level activity in an adaptive and robust way can be useful in many contexts (see, e.g., Section 2.5). Therefore, it came natural to ask whether, to what extent, and how emergence could be *managed* (also, *steered*) and *engineered* [SPT06; NZ15; MDT18; RJ18], i.e., developed in a principled way. This is a central theme of this thesis. In Chapter 5, a formal framework and toolchain for "engineering emergence", called *aggregate programming*, is presented. It provides a linguistic approach for compositionally specifying global (i.e., emergent) behaviour—ultimately implemented through repeated context perception, computation, and neighbourhood-based interaction by a set of situated devices.

2.4 Pervasive and Ubiquitous Computing

Almost three decades ago, in his seminal paper [Wei91], Weiser foresaw the IoT and CPS trends:

> *Specialized elements of hardware and software, connected by wires, radio waves and infrared, will be so ubiquitous that no one will notice their presence.*

and identified *ubiquitous computing* as the 3rd generation of computing (after mainframe and personal computing), characterised by an embodied virtuality where countless computers are pushed into the background, fostering cyber-physical integration, transparency, and usability:

> *When almost every object either contains a computer or can have a tab attached to it, obtaining information will be trivial [...]*
>
> *[...] machines that fit the human environment instead of forcing humans to enter theirs [...]*

ubiquitous vs. pervasive computing

Ubiquitous computing and pervasive computing are similar and related ideas. Term "ubiquitous" means "present everywhere", whereas "pervasive" means "existing in or spreading through every part of something". Though, terminologically, distinctions may be artificial, ubiquitous computing is generally intended as aiming to *transparently* provide computational services everywhere through embedding (i.e., a sort of "disappearing computing" [SN05]), whereas pervasive computing typically refers to having computers "touch" any aspect of human life, not necessarily in an invisible way.

Other perspectives or concepts related to ubiquitous computing include *calm computing* [WB96],

> *A calm technology will move easily from the periphery of our attention, to the center, and back.*

as well as *invisible computing* [Nor99; Bor00] and, most notably, *ambient computing/intelligence* [Duc+01; Sad11].

2.4.1 Ambient intelligence

ambient intelligence

In the survey [Sad11], ambient intelligence is described as

> *the vision of a future in which environments support the people inhabiting them. This envisaged environment is unobtrusive, interconnected, adaptable, dynamic, embedded and intelligent. In this vision the traditional computer input and output media disappear. Instead processors and sensors are integrated in everyday objects.*

In [AM03], Aarts and Marzano suggest that ambient intelligence is characterised by the following features: *(1) embedded*, *(2) context-aware*, *(3) personalised*, *(4) adaptive*, and *(5) anticipatory*. This is quite related to the notion of spatial computing, which is covered in Chapter 4.

Another related term is *situated computing*, which is described in [HNBR+97] as an approach that

> *concerns the ability of computing devices to detect, interpret and respond to aspects of the user's local environment.*

The point of all these notions is to recognise the prominent role of the environment and the context for seamless service provisioning.

2.4.2 Context-aware computing

context-aware computing

Context-aware computing is about using context (i.e., any information available) to characterise the situation in order to provide (better) services to users. A survey regarding the engineering of context-aware systems is provided in [AAC16]. The features of a context-aware system can be classified into [AAC16]: *(1) presenting context* to stakeholders; *(2) active/passive service execution*[1]; *(3) active/-passive service configuration*; and *(4) mapping context to information* for later retrieval. The lifecycle of context information [AAC16] involves acquisition (from potentially heterogeneous context sources), modelling, reasoning, and (real-time) dissemination. Regarding the engineering of context-aware systems [AAC16], traditional paradigms include the object-oriented, aspect-oriented, feature-oriented, and service-oriented ones, with agent-oriented and context-oriented [HCN08] emerging.

context-oriented programming

Context-oriented programming In the survey [SGP12] on context-oriented software engineering, *context-oriented programming (COP)* is described as a paradigm that *"tackles the issue of developing context-aware systems at the language-level, introducing ad hoc language abstractions to manage adaptations modularization and their dynamic activation"*. In other words, COP focuses on providing mechanisms for implementing software that behaves differently according to the *context*, which is loosely defined as *"any information which is computationally accessible"* [HCN08]. Usually, this is achieved by organising context-aware

[1] The active-passive degree refers to the extent of user involvement: active execution or configuration refers to autonomous activity by the system, whereas passivity means user involvement is needed.

code into modular units called *behavioural variations* [HCN08] which are to be dynamically activated or deactivated according to context changes. Commonly, these take the form of named *layers* [CH05] that group related partial method definitions. So, a layer represents a certain aspect of a context, and the current context is given by the set of all currently active layers.

COP fosters *adaptation* by dynamically supporting change of behaviour (possibly, of multiple system components) based on change of context. The prominence of context in emerging situated, distributed computing scenarios makes the COP vision and, in general, the reification of context as a first-class abstraction, a significant research perspective. In [AHR08], the authors motivate COP for ubiquitous computing.

2.5 Collective Computing

collectives

Term "collective" derives from Latin *collectivus*, in turn from *collectus*, past participle of *colligere*, which means "to gather together". So, a collective is an entity that gather multiple congeneric elements (i.e., belonging to the same genus, namely, of related nature) together. In other words, a *collective* is a group of similar individuals or entities that share something (e.g., a goal, a reason for unity, an environment, an interface). For instance, a group of people or agents is a collective, whereas a gathering of radically different entities such as cells, rivers, and books is (intuitively) not.

2.5.1 Computational collective and swarm intelligence

collective intelligence

In psychological science, *collective intelligence* can be defined as the property of a group of individuals that emerges from bottom-up and top-down processes and that allows the group to perform a wide variety of tasks [WAM15]. One of the goals in this research areas is to identify a *c factor* for general collective intelligence which would be the analogue of the *g factor* for general individual intelligence, e.g., enabling prediction of group performance [WAM15].

computational collective intelligence

In this thesis, we are partly concerned with *computational collective intelligence (CCI)*, i.e., *"the form of intelligence that emerges from the collaboration and*

competition of many individuals (artificial and/or natural)" [NKC09]. The CCI research area covers methodological, theoretical, and practical aspects of CCI and is strictly related to other fields such as computational intelligence (including swarm intelligence), semantic web, social network analysis, multi-agent systems (*"as a computational and modeling paradigm especially tailored to capture the nature of CCI emergence in populations of autonomous individuals"* [NKC09]), and theories of group decision-making and consensus.

swarm intelligence

Term *swarm intelligence (SI)* was first introduced by Beni et al. in the context of cellular robotics [BW89]; it refers to *"a kind of problem-solving ability that emerges in the interactions of simple information-processing units"* [Ken06]. In swarm-based systems, organisation and functionality emerge from decentralised interaction of (usually simple) agents. SI is inspired by the way many biological systems work [Bon+99], such as societies or groups of ants, bees, fireflies, bats, etc [PL11]. Historically, the two main SI approaches were *ant colony optimisation* and *particle swarm optimisation* [PL11], paving the path for many other techniques. SI is considered as a part of *computational intelligence* [Eng07], together with other bio-inspired fields (cf., *bio-inspired computing* [Kar16] and *organic computing* [MSSU11]) like artificial neural networks, evolutionary computation, artificial immune systems, and fuzzy systems. A relevant field related to SI is *swarm robotics* [Bra+13]. A *swarm* provides some interesting advantages over centralised systems [Ben09; Bra+13], namely *(i) flexible functionality*, through capabilities that go beyond those of individuals; *(ii) robustness*, through redundancy of components and tolerance of the loss of few individuals; *(iii) economy*, through simplicity of individuals, which can be mass produced, interchanged, and easily disposed; and *(iv) scalability*, by splitting work and communication over a large number of elements. Potential issues may include *(i) efficiency*, as consensus and decision-making may negatively affect group reactivity; and *(ii) consequences of emergence* (see Section 2.3.2), such as the difficulty to obtain hard guarantees or to safely steer the system.

computational intelligence

swarm robotics

2.5.2 Collective adaptive systems

collective adaptive system

By definition, a *collective adaptive system (CAS)* is a *system* – i.e., a set of interacting entities – that is, crucially:

- *collective*—i.e., consists of a (possibly very large) number of congeneric individuals; and
- *adaptive*—i.e., it is able to change its control rules as a consequence of internal or external changes.

An example of *non-collective* (possibly adaptive) system is one in which individuals are of entirely different nature or are driven by unrelated goals which have no coherence at the global level. An example of collective but *non-adaptive* system is a cellular automaton, since it is driven by fixed rules.

As for MASs, key for a characterisation of a collective system is the definition of at least two *levels*—the *micro* level of the units, and the *macro* level of the whole. Of course, a collective may consist of multiple (transient or permanent) sub-collectives and may be part of super-collectives. Like for systems, boundaries and containment of collectives essentially depend on the perspective, for both analysis and synthesis.

As for MASs, CASs often consist of *autonomous* agents. Autonomy, generally intended as the ability of "self-government", is a complex notion that can have multiple characterisations and may exist in degrees (i.e., it is not a black or white feature). Autonomy is usually a source for *unpredictability*, as behaviours and responses may vary depending on an individual's self-determination. The challenge in MASs and CASs is to promote *collaboration* between components, such that they can together carry out tasks that none of them, as a single entity, would be able to do [CPZ11]. This is especially hard in *competitive* settings, where each component is fully *self-interested*, but has to interact with other components to achieve its goal [Kep+17]; this may involve *negotiation* and *trust*.

The other key source for unpredictability is the *environment*, for it usually has complex dynamics. CASs are essentially *situated*, i.e., they are made of components that are immersed into some (logical and/or physical) environment and are engaged in non-trivial interactions with it. Indeed, such (eco-)systems are *adaptive* just because they need to *evolve* in order to respond to changes in the environment or in the input patterns.

By a structural point of view, a CAS may exhibit various and possibly dynamic structures, often constructed and sustained through a *self-organisation* process, i.e., a robust, internal process by which internal order is continually sought, often

in an emergent way [DWH04]. Indeed, the presence of several (autonomous) components requires an appropriate organisation to be enforced so as to assign roles and responsibilities to the components themselves [ZJW05] and promote ordered development of collective behaviour. Tightly related to organisation is *coordination*, namely, the enaction of rules to constrain interaction.

Also, crucially, CASs often feature *emergence*, whereby macro-level properties and behaviours spring out from decentralised, micro-level activity. The key point is that the global properties arising from the interaction of the parts cannot be easily traced back to properties and behaviours of the parts. This poses significant challenges regarding understanding, designing, and controlling CASs. However, emergence is typical because CASs often consist of partially autonomous components and feature *decentralised control*—i.e., there is no single component which governs the collective behaviour. Decentralisation is fundamental for systems to *scale* (as adding more components is less susceptible to overloading functionality) and achieve *robustness* (as functionality is not confined to few, critical components). Indeed, decentralisation of control, non-synchronised operation, and opportunistic interaction are often essential in certain contexts to deal with the scale and changes in both the system structure and environment.

CASs exist in the world, in our minds (cf., mental models), in theory (cf., mathematical models), and in software [MK+15]. The nature has indeed been a great source of inspiration for mechanisms used by engineers to endow artificial systems with features like self-organisation and *resilience* (i.e., the ability to recover from failure). CAS-oriented features are especially useful in scenarios characterised by *high dynamicity* (i.e., where the environment or the inputs have complex dynamics), *openness* (i.e., where components can dynamically enter or leave the system), and *(very) large scale*.

2.6 Final Remarks

This chapter introduces a set of concepts related to collective adaptive systems research. It does so by describing multiple inter-related perspectives and corresponding research threads: multi-agent systems (with emphasis on coordination and organisations), autonomic computing (with emphasis on self-* properties),

pervasive/ubiquitous computing (with emphasis on context-awareness and ambient intelligence), and collective computing (with emphasis on collective intelligence). As we will see, this thesis is particularly concerned with the problem of modelling the computational behaviour of situated collectives that need to self-adapt to the environment and coordinate to resiliently produce interesting global features.

References

[AAC16] Unai Alegre, Juan Carlos Augusto, and Tony Clark. "Engineering context-aware systems and applications: A survey". In: *Journal of Systems and Software* 117 (2016), pp. 55–83.

[Abo16] Gregory D Abowd. "Beyond weiser: From ubiquitous to collective computing". In: *Computer* 49.1 (2016), pp. 17–23.

[AHR08] Malte Appeltauer, Robert Hirschfeld, and Tobias Rho. "Dedicated programming support for context-aware ubiquitous applications". In: *2008 The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. IEEE. 2008, pp. 38–43.

[AM03] Emile Aarts and Stefano Marzano. *The new everyday: Views on ambient intelligence*. 010 Publishers, 2003.

[And+13] S Anderson, N Bredeche, AE Eiben, G Kampis, and MR van Steen. "Adaptive collective systems: herding black sheep". In: (2013).

[AP08] Alexander Artikis and Jeremy Pitt. "Specifying open agent systems: A survey". In: *International Workshop on Engineering Societies in the Agents World*. Springer. 2008, pp. 29–45.

[Ash61] W Ross Ashby. *An introduction to cybernetics*. Chapman & Hall Ltd, 1961.

[BBF09] Gordon Blair, Nelly Bencomo, and Robert B France. "Models@ run. time". In: *Computer* 42.10 (2009), pp. 22–27.

[Ben09] Gerardo Beni. "Swarm intelligence". In: *Encyclopedia of Complexity and Systems Science* (2009), pp. 1–32.

[Ber69] Ludwig von Bertalanffy. *General system theory : foundations, development, applications*. New York: G. Braziller, 1969. URL: `http://opac.inria.fr/record=b1078794`.

[BGS19] Nelly Bencomo, Sebastian Götz, and Hui Song. "Models@run.time: a guided tour of the state of the art and research challenges". In: *Software & Systems Modeling* 18.5 (2019), pp. 3049–3082. ISSN: 1619-1374. DOI: `10.1007/s10270-018-00712-x`.

[BHS06] Olivier Boissier, Jomi Fred Hübner, and Jaime Simão Sichman. "Organization oriented programming: From closed to open organizations". In: *International Workshop on Engineering Societies in the Agents World*. Springer. 2006, pp. 86–105. DOI: 10.1007/978-3-540-75524-1_5.

[Bon+99] Eric Bonabeau, Directeur de Recherches Du Fnrs Marco, Marco Dorigo, Guy Théraulaz, Guy Theraulaz, et al. *Swarm intelligence: from natural to artificial systems*. 1. ¿ 8000 cits. Oxford university press, 1999.

[Bor00] Gaetano Borriello. "The challenges to invisible computing". In: *Computer* 33.11 (2000), pp. 123–125.

[Bra+13] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. "Swarm robotics: a review from the swarm engineering perspective". In: *Swarm Intelligence* 7.1 (2013), pp. 1–41.

[BW89] G Beni and J Wang. "Swarm Intelligence in Cellular Robotic Systems, Proceed. NATO Advanced Workshop on Robots and Biological Systems, Tuscany, Italy, June 26-30". In: *Y.: NATO* (1989).

[BY02] Yaneer Bar-Yam. "General features of complex systems". In: *Encyclopedia of Life Support Systems (EOLSS), UNESCO, EOLSS Publishers, Oxford, UK* 1 (2002).

[Cao+13] Yongcan Cao, Wenwu Yu, Wei Ren, and Guanrong Chen. "An overview of recent progress in the study of distributed multi-agent coordination". In: *IEEE Transactions on Industrial informatics* 9.1 (2013), pp. 427–438. DOI: 10.1109/TII.2012.2219061.

[Cas98] Cristiano Castelfranchi. "Modelling social action for AI agents". In: *Artificial intelligence* 103.1-2 (1998), pp. 157–182.

[CH05] Pascal Costanza and Robert Hirschfeld. "Language constructs for context-oriented programming: an overview of ContextL". In: *Proceedings of the 2005 symposium on Dynamic languages*. ACM. 2005, pp. 1–10.

[Che00] Peter Checkland. "Systems thinking, systems practice: includes a 30-year retrospective". In: *Journal-Operational Research Society* 51.5 (2000), pp. 647–647.

[CIMV11] F Stuart Chapin III, Pamela A Matson, and Peter Vitousek. *Principles of terrestrial ecosystem ecology*. Springer Science & Business Media, 2011.

[CPZ11] Giacomo Cabri, Mariachiara Puviani, and Franco Zambonelli. "Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles". In: *2011 International Conference on Collaboration Technologies and Systems, CTS 2011, Philadelphia, Pennsylvania, USA, May 23-27, 2011*. 2011, pp. 508–515.

[Duc+01] Ken Ducatel, Union européenne. Technologies de la société de l'information, Union européenne. Institut d'études de prospectives technologiques, and Union européenne. Société de l'information conviviale. "Scenarios for ambient intelligence in 2010". In: (2001).

[DWH04] Tom De Wolf and Tom Holvoet. "Emergence versus self-organisation: Different concepts but promising when combined". In: *International workshop on engineering self-organising applications*. Springer. 2004, pp. 1–15.

[Eng07] Andries P Engelbrecht. *Computational intelligence: an introduction*. John Wiley & Sons, 2007.

[Est+01] Marc Esteva, Juan-Antonio Rodriguez-Aguilar, Carles Sierra, Pere Garcia, and Josep L Arcos. "On the formal specification of electronic institutions". In: *Agent mediated electronic commerce*. Springer, 2001, pp. 126–147.

[Etz64] Amitai Etzioni. "Modern organizations, 1964". In: *NJ: Englewood Cliffs* (1964).

[FG96] Stan Franklin and Art Graesser. "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents". In: *International Workshop on Agent Theories, Architectures, and Languages*. Springer. 1996, pp. 21–35.

[FW99] Jacques Ferber and Gerhard Weiss. *Multi-agent systems: an introduction to distributed artificial intelligence*. Vol. 1. Addison-Wesley Reading, 1999.

[Gil08] Helen Gill. "From vision to reality: cyber-physical systems". In: *HCSS national workshop on new research directions for high confidence transportation CPS: automotive, aviation, and rail*. 2008.

[Gun+14] Volkan Gunes, Steffen Peter, Tony Givargis, and Frank Vahid. "A survey on concepts, applications, and challenges in cyber-physical systems." In: *KSII Transactions on Internet & Information Systems* 8.12 (2014).

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Marius Nierstrasz. "Context-oriented programming". In: *Journal of Object technology* 7.3 (2008), pp. 125–151.

[HL04] Bryan Horling and Victor Lesser. "A survey of multi-agent organizational paradigms". In: *The Knowledge engineering review* 19.4 (2004), pp. 281–316.

[HNBR+97] Richard Hull, Philip Neaves, James Bedford-Roberts, et al. *Towards situated computing*. Hewlett Packard Laboratories, 1997.

[HS06] Michael G Hinchey and Roy Sterritt. "Self-managing software". In: *Computer* 39.2 (2006), pp. 107–109.

[HSB07] Jomi F Hübner, Jaime S Sichman, and Olivier Boissier. "Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels". In: *International Journal of Agent-Oriented Software Engineering* 1.3/4 (2007), pp. 370–395. DOI: 10.1504/IJAOSE.2007.016266.

[HW11] Christopher D Hollander and Annie S Wu. "The current state of normative agent-based systems". In: *Journal of Artificial Societies and Social Simulation* 14.2 (2011), p. 6.

[Kar16] Arpan Kumar Kar. "Bio inspired computing–a review of algorithms and scope of applications". In: *Expert Systems with Applications* 59 (2016), pp. 20–32.

[KC03] Jeffrey O Kephart and David M Chess. "The vision of autonomic computing". In: *Computer* 1 (2003), pp. 41–50.

[Ken06] James Kennedy. "Swarm intelligence". In: *Handbook of nature-inspired and innovative computing*. ¿ 9600 cits. Springer, 2006, pp. 187–219.

[Kep+17] Jeffrey O Kephart, Ada Diaconescu, Holger Giese, Anders Robertsson, Tarek Abdelzaher, Peter Lewis, Antonio Filieri, Lukas Esterle, and Sylvain Frey. "Self-adaptation in collective self-aware computing systems". In: *Self-Aware Computing Systems*. Springer, 2017, pp. 401–435.

[Mae87] Pattie Maes. "Concepts and experiments in computational reflection". In: *ACM Sigplan Notices*. Vol. 22. 12. ACM. 1987, pp. 147–155.

[Mai98] Mark W. Maier. "Architecting principles for systems-of-systems". In: *Systems Engineering* 1.4 (1998), pp. 267–284. DOI: 10.1002/(sici)1520-6858(1998)1:4<267::aid-sys3>3.0.co;2-d. URL: https://doi.org/10.1002/(sici)1520-6858(1998)1:4<267::aid-sys3>3.0.co;2-d.

[MDT18] Saurabh Mittal, Saikou Diallo, and Andreas Tolk. *Emergent behavior in complex systems engineering: a modeling and simulation approach*. John Wiley & Sons, 2018.

[MK15] George E. Mobus and Michael C. Kalton. *Principles of Systems Science*. Springer, 2015. ISBN: 9781493919208.

[MK+15] George E Mobus, Michael C Kalton, et al. *Principles of systems science*. Springer, 2015.

[MSSU11] Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer. *Organic computing—a paradigm shift for complex systems*. Springer Science & Business Media, 2011.

[Mul12] Robert Mullins. "The EternalS Roadmap–Defining a Research Agenda for Eternal Systems". In: *International Workshop on Eternal Systems*. Springer. 2012, pp. 135–147.

[NKC09] Ngoc Thanh Nguyen, Ryszard Kowalczyk, and Shyi-Ming Chen. "Computational Collective Intelligence. Semantic Web, Social Networks and Multiagent Systems". In: *Conference proceedings ICCCI*. Springer. 2009, p. 269.

[NLJ96a] Hyacinth S Nwana, Lyndon C Lee, and Nicholas R Jennings. "Coordination in software agent systems". In: *British Telecom Technical Journal* 14.4 (1996), pp. 79–88.

[NLJ96b] Hyacinth S Nwana, Lyndon C Lee, and Nicholas R Jennings. "Coordination in software agent systems". In: *British Telecom Technical Journal* 14.4 (1996), pp. 79–88.

[NN95] Grégoire Nicolis and G Nicolis. *Introduction to nonlinear science*. Cambridge University Press, 1995.

[Nor99] Donald A Norman. *The invisible computer: why good products can fail, the personal computer is so complex, and information appliances are the solution*. MIT press, 1999.

[NZ15] Victor Noël and Franco Zambonelli. "Methodological Guidelines for Engineering Self-organization and Emergence". In: *Software Engineering for Collective Autonomic Systems - The ASCENS Approach*. 2015, pp. 355–378. DOI: 10.1007/978-3-319-16310-9_10. URL: https://doi.org/10.1007/978-3-319-16310-9_10.

[Ode02] James Odell. "Objects and agents compared". In: *Journal of object technology* 1.1 (2002), pp. 41–53.

[Omi+04] Andrea Omicini, Alessandro Ricci, Mirko Viroli, Cristiano Castelfranchi, and Luca Tummolini. "Coordination artifacts: Environment-based coordination for intelligent agents". In: *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*. IEEE Computer Society. 2004, pp. 286–293.

[OO03] Andrea Omicini and Sascha Ossowski. "Objective versus subjective coordination in the engineering of agent systems". In: *Intelligent information agents*. Springer, 2003, pp. 179–202.

[Ore+99] Peyman Oreizy, Michael M Gorlick, Richard N Taylor, Dennis Heimhigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S Rosenblum, and Alexander L Wolf. "An architecture-based approach to self-adaptive software". In: *IEEE Intelligent Systems and Their Applications* 14.3 (1999), pp. 54–62.

[ORV08] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. "Artifacts in the A&A meta-model for multi-agent systems". In: *Autonomous agents and multi-agent systems* 17.3 (2008), pp. 432–456.

[Ott04] Julio M Ottino. "Engineering complex systems". In: *Nature* 427.6973 (2004), p. 399.

[Par97] H Van Dyke Parunak. ""Go to the ant": Engineering principles from natural multi-agent systems". In: *Annals of Operations Research* 75 (1997), pp. 69–101.

[PL11] Rafael S Parpinelli and Heitor S Lopes. "New inspirations in swarm intelligence: a survey". In: *International Journal of Bio-Inspired Computation* 3.1 (2011), pp. 1–16.

[Pla+07] Eric Platon, Marco Mamei, Nicolas Sabouret, Shinichi Honiden, and H Van Dyke Parunak. "Mechanisms for environments in multi-agent systems: Survey and opportunities". In: *Autonomous Agents and Multi-Agent Systems* 14.1 (2007), pp. 31–47.

[RA18] Sutton Richard and Barto Andrew. *Reinforcement learning: an introduction*. MIT Press, 2018.

[RG95] Anand S. Rao and Michael P. Georgeff. "BDI Agents: From Theory to Practice". In: *Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*. 1995, pp. 312–319.

[RJ18] Larry B Rainey and Mo Jamshidi. *Engineering emergence: A modeling and simulation approach*. CRC Press, 2018.

[Sad11] Fariba Sadri. "Ambient intelligence: A survey". In: *ACM Computing Surveys (CSUR)* 43.4 (2011), p. 36.

[SGK05] Giovanna Di Marzo Serugendo, Marie-Pierre Gleizes, and Anthony Karageorgos. "Self-organization in multi-agent systems". In: *The Knowledge Engineering Review* 20.2 (2005), pp. 165–189. DOI: 10.1017/S0269888905000494.

[SGP12] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. "Context-oriented programming: A software engineering perspective". In: *Journal of Systems and Software* 85.8 (2012), pp. 1801–1817. ISSN: 01641212. DOI: 10.1016/j.jss.2012.03.024.

[Sil12] Hillary Sillitto. "4.3. 2 Integrating Systems Science, Systems Thinking, and Systems Engineering: understanding the differences and exploiting the synergies". In: *INCOSE International Symposium*. Vol. 22. 1. Wiley Online Library. 2012, pp. 532–547.

[Sin+12] A Singer, H Sillitto, J Bendz, G Chroust, D Hybertson, HB Lawson, J Martin, R Martin, M Singer, and T Takaku. "The Systems Praxis Framework". In: *Proceedings of the IFSR Conversation* (2012).

[Smi80] Reid G Smith. "The contract net protocol: High-level communication and control in a distributed problem solver". In: *IEEE Transactions on computers* 12 (1980). ¿4900 cits, pp. 1104–1113.

[SN05] Norbert Streitz and Paddy Nixon. "The disappearing computer". In: *Communications-ACM* 48.3 (2005), pp. 32–35.

[SPT06] Susan Stepney, Fiona Polack, and Heather R. Turner. "Engineering Emergence". In: *11th International Conference on Engineering of Complex Computer Systems (ICECCS 2006), 15-17 August 2006, Stanford, California, USA*. 2006, pp. 89–97. DOI: 10.1109/ICECCS.2006.55. URL: http://doi.ieeecomputersociety.org/10.1109/ICECCS.2006.55.

[ST09] Mazeiar Salehie and Ladan Tahvildari. "Self-adaptive software: Landscape and research challenges". In: *ACM transactions on autonomous and adaptive systems (TAAS)* 4.2 (2009), p. 14.

[Tom+14] Sven Tomforde, Jörg Hähner, Hella Seebach, Wolfgang Reif, Bernhard Sick, Arno Wacker, and Ingo Scholtes. "Engineering and Mastering Interwoven Systems". In: *ARCS 2014 - 27th International Conference on Architecture of Computing Systems, Workshop Proceedings, February 25-28, 2014, Luebeck, Germany, University of Luebeck, Institute of Computer Engineering*. 2014, pp. 1–8. URL: http://ieeexplore.ieee.org/document/6775093/.

[Vir+07] Mirko Viroli, Tom Holvoet, Alessandro Ricci, Kurt Schelfthout, and Franco Zambonelli. "Infrastructures for the environment of multiagent systems". In: *Autonomous Agents and Multi-Agent Systems* 14.1 (2007), pp. 49–60. DOI: 10.1007/s10458-006-9001-6. URL: https://doi.org/10.1007/s10458-006-9001-6.

[VR04] Marc HV Van Regenmortel. "Reductionism and complexity in molecular biology". In: *EMBO reports* 5.11 (2004), pp. 1016–1020.

[WAM15] Anita Williams Woolley, Ishani Aggarwal, and Thomas W Malone. "Collective intelligence and group performance". In: *Current Directions in Psychological Science* 24.6 (2015), pp. 420–424.

[WB96] Mark Weiser and John Seely Brown. "Designing calm technology". In: *PowerGrid Journal* 1.1 (1996), pp. 75–85.

[Wea48] Warren Weaver. "Science and Complexity". In: *American Scientist* 36.536–544 (1948).

[Wei91] Mark Weiser. "The Computer for the 21 st Century". In: *Scientific american* 265.3 (1991), pp. 94–105.

[Wey+12] Danny Weyns, M Usman Iftikhar, Didac Gil De La Iglesia, and Tanvir Ahmad. "A survey of formal methods in self-adaptive systems". In: *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*. ACM. 2012, pp. 67–79.

[Wie65] Norbert Wiener. *Cybernetics or Control and Communication in the Animal and the Machine*. Vol. 25. MIT press, 1965.

[WOO07] Danny Weyns, Andrea Omicini, and James Odell. "Environment as a first class abstraction in multiagent systems". In: *Autonomous agents and multi-agent systems* 14.1 (2007), pp. 5–30.

[Woo09] Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[Woo97] Michael Wooldridge. "Agent-based software engineering". In: *IEE Proceedings-software* 144.1 (1997), pp. 26–37.

[ZJW05] Franco Zambonelli, Nicholas R Jennings, and Michael Wooldridge. "Multi-agent systems as computational organizations: the Gaia methodology". In: *Agent-oriented methodologies*. IGI Global, 2005, pp. 136–171.

Chapter 3

Distributed Computing and Coordination

Order. Let all your things have their places; let each part of your business have its time.

Benjamin Franklin

Contents

This thesis is mostly concerned with *distributed computing*. A *distributed system* can be defined as a computer system comprising multiple *processors* exchanging *messages* over a *communication network* [Gar02]. Having distributed components inherently leads to concurrency, lack of global clock, and independent (and often frequent) failure or unavailability of components [CDK05]—with corresponding implications. Therefore, the research area of distributed computing aims

distributed computing

at addressing the uncertainty emerging when the spatio-temporal unit of systems is lost, with issues including handling communication, naming, synchronisation of activities, consistency, replication, and failure [TVS07]. While *distributed algorithms* [Lyn96] are used to perform specific input/output tasks in distributed systems (e.g., related to consensus, leader election, snapshotting, etc.), *coordination* is used to support functionality by *"managing dependencies between activites"* [MC94].

In fact, in interactive systems, there are at least two independent dimensions: *computation* and *interaction*. Coordination consists of ruling the "interaction space", i.e., determining and fostering admissible interaction. Ciancarini describes a meta-model for coordination systems [Cia96] which consists of three classes of entities:

1. *coordinables* — the entities to coordinate;
2. *coordination media* — the abstractions that support coordination; and
3. *coordination laws* — the rules constraining coordinables, the media, and their interaction.

Then, *coordination models* define these elements and *coordination languages* provide a way to express them in a formal way by defining syntax and semantics of the primitives of interaction. According to [PA98], coordination models and languages can be classified into :

1. *data-driven* (or *task-oriented*) — where interaction is based on information exchange (e.g., through a shared dataspace);
2. *control-driven* (or *process-oriented*) — where events and not what data is handled within a process are of interest to the coordination media, which consists of input and output communication ports connecting processes (considered as black boxes).

In this chapter, as outlined in Figure 3.2, we review and discuss the conceptual, but also technical and technological, path that has brought traditional coordination models for concurrent systems, step-by-step to address the complexity of self-organising, large-scale deployed systems.

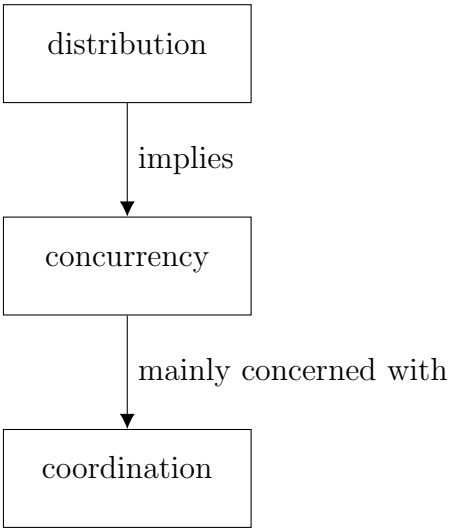


Figure 3.1: Coordination is the main theme of distributed systems.

3.1 Concurrency Theory, Processes, and Services

concurrency theory

As distributed systems are essentially concurrent, tools for modelling and verifying them are provided in the context of *concurrency theory* [BG06], i.e., the mathematical study of interacting and simultaneously evolving processes. This field originated in the 1960s with the pioneering work of Carl Petri that launched *Net theory* [Pet66] as a framework for specifying and analysing communicating concurrent systems. Another main class of approaches is given *process algebras* or *process calculi*, whose research line originated between the end of the 1970s and the beginning of 1980s with the independent formulation of the three classic formal languages: Hoare's *Communicating Sequential Processes (CSP)* [Hoa78], Milner's *Calculus of Communicating Systems (CCS)* [Mil89], and the *Algebra of Communicating Processes (ACP)* [BK84] by Bergstra and Klop. Historical treatments of the development of process algebras can be found, e.g., in [Bae05; AG05].

A significant representative of this research line is the π*-calculus* [Mil99], which models concurrent computation as a set of processes that interact by reading from and writing to shared channels. With respect to its predecessors, the π-calculus also attempts to model *mobility*: it does so by supporting dynamic reconfiguration

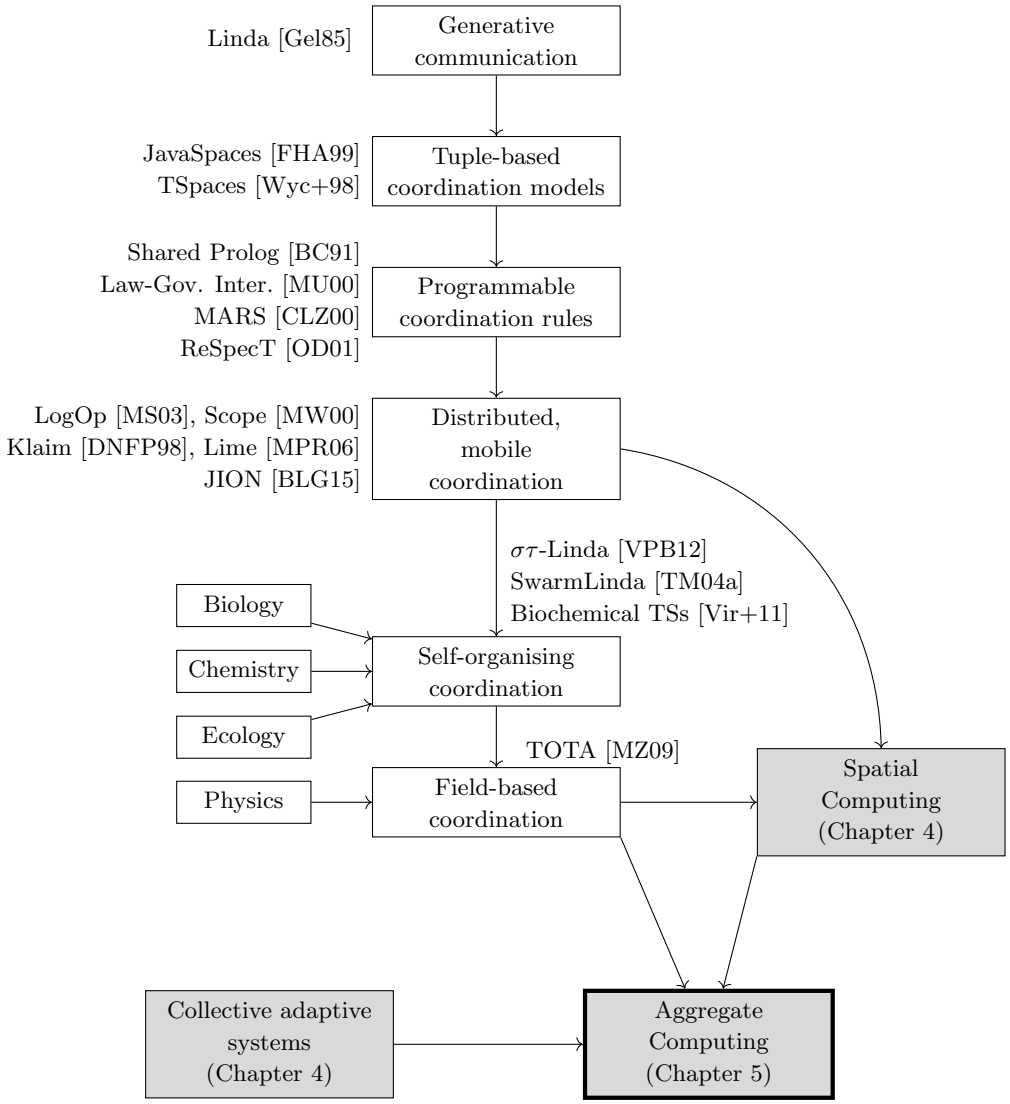


Figure 3.2: Overview of research threads leading from coordination to field calculus and aggregate computing, with some representative bibliographic references. This summary—by no means exhaustive—provides key highlights mainly from the perspective of Chapter 2.

of the system topology by exchanging channel names over channels themselves. In this framework, an important enquiry concerns semantic equivalence between two processes. An equivalence notion is given by *bisimilarity*, which captures whether two processes are able to mimic one another; a procedure for checking bisimilarity is *bisimulation*. Another relevant related work is *bigraphs* [Mil09], a formalism for ubiquitous systems where nested graphs are used to model the notions of agent, locality, connectivity, motion, and interaction.

service-oriented computing

Another, related paradigm for describing communicating, distributed systems is *service-oriented computing (SOC)* [BZ07]. In SOC, applications are built through a set of collaborating *services*, i.e., *"autonomous platform-independent elements that can be described, published and discovered using interoperable standards"* [BZ07]. This approach took momentum in the 1990s with the advent of *Web Services*, a set of technologies to building applications as sets of services interacting through Web standards. The main standards include *WSDL (Web Services Description Language)* for describing the interfaces of services, *Simple Object Access Protocol (SOAP)* for describing message formats and exchange patterns, and *Universal Description, Discovery, and Integration (UDDI)* for service discovery. Regarding *service composition* [LDB15], two main styles exist: *orchestration*, where coordination is supported by a centralised entity (called the *orchestrator*) from a local perspective, and *choreography* [Pel03], whereby decentralised interaction among multiple parties unfolds from an interaction protocol defined by a global perspective.

3.2 Shared Dataspace Coordination

Coordination models are rooted in the idea that interaction among multiple, independent, and autonomous software systems (e.g., processes, components, and so on, somewhat generically called *agents* henceforth) could be conceived and designed as a space orthogonal to pure computation. Historically, many coordination models reify this idea into a concept of *shared dataspace*, working as a whiteboard, where processes of a parallel computing system can write and read information [Cor91], enabling so-called *generative communication* [Gel85]. In generative communication, the "life" of generated data is independent of the "life" of

generative communication

the generator; this, crucially, enables *space decoupling* (two processes do not need to be spatially co-located to interact), *time decoupling* (two processes do not need to be temporally co-located to interact).

3.2.1 Generative communication

Linda

tuple-based coordination

The *Linda coordination model* [Gel85] is broadly recognised as the ancestor of a number of approaches to generative communication falling under the umbrella of *tuple-based coordination models*. The foundational idea of Linda was to have processes (on a centralised system) share information and synchronise by writing and retrieving, with a suspensive semantics (the requester is blocked until the query is satisfiable), data in the form of an ordered collection of possibly heterogeneous knowledge chunks, i.e., *tuples*, from a shared (tuple-)space. Such data could be retrieved *associatively*, by querying through partial representations of the structure and content matching the desired piece of data (*tuple template*). The consequence is twofold: *(i)* decoupling in communication is strongly promoted, since no information about the sender, the space itself, and the tuple insertion time is required in order for communication to happen; and *(ii)* coordination is still possible in environments where information is vague, incomplete, inaccurate, or not entirely specified, due to the possibility of synchronising over a partial representation of knowledge.

3.2.2 Programmable coordination rules

tuple spaces

The vision of tuple-based coordination as a shared knowledge repository used for agent coordination is further promoted by logical *tuple-space models*, where software agents coordinate through first-order logic tuples, and tuple spaces can be programmed as first-order logic theories. A prominent example of such approach is *Shared Prolog* [BC91], a framework for writing multi-processor Prolog systems. More generally, this view promotes the idea of equipping the shared space with some form of "intelligence", e.g., in the form of an application logic that can manipulate data in the shared space and the way that it can be accessed. Several Linda-inspired approaches tackle this issue by enabling programmability at the tuple-space level in order to express rules of coordination, and hence, pushing

forward a notion of *expressiveness of the coordination media* [Bus+01]. Examples include the following.

Law-Governed Interaction [MU00] — It structures the coordination logic within groups of agents by explicit "rules of engagement".

MARS (Mobile Agent Reactive Spaces) [CLZ00] — In MARS, tuple spaces can be programmed with stateful "reaction objects" triggered upon access patterns

ReSpecT (Reaction Specification Tuples) [OD01] — In this framework, logic specification tuples map events to transactional sequences of reactions, which are primitive invocations of logic-based computations.

3.3 Distributed coordination

Many of the approaches outlined before, however, do not explicitly focus on distributed systems, but on the coordination of centralised local components. As software components become spread across the system network, so multiple tuple spaces can be distributed across the system environment, enabling distributed coordination abstractions, featuring mechanisms for event-based interactions, timing, and advanced data representation. This is the case with industrial systems like *JavaSpaces* [FHA99], an API for distributed coordination through persistent, shared spaces of objects, and *TSpaces* [Wyc+98], which combines Linda-like spaces with asynchronous messaging.

Some middlewares take the approach a step further, by dealing with location and mobility, and enabling expression of dynamic environment topologies in a distributed setting, thus paving the way towards application of coordination models to pervasive computing system scenarios. Indeed, tuple-based coordination approaches have been also used in the context of *peer-to-peer (P2P)* and *mobile ad-hoc networks (MANETs)*, where there is no pre-existing infrastructure and devices interact opportunistically via short-range wireless technology. These scenarios share various characteristics – mobility, dynamicity, locality, openness – and can be considered a special case of physical deployment of situated multi-

agent systems [BMS02], where the aforementioned features are not just issues but also opportunities that can be exploited to construct collective intelligence (cf., naturally-inspired systems like *SwarmLinda* [TM04b]). Systems for tuple-based coordination in these contexts usually come with a middleware, dealing with certain issues related to distribution and mobility, and extensions to the basic tuple-space model and Linda language to support specific aspects, e.g., related to location management. Moreover, since networks of devices can be physically situated (cf., MANETs), it comes natural to consider nodes as representatives of space-time locations, and hence extend coordination models for such contexts with first-class space and time abstractions (as covered in Chapter 4).

JION (JavaSpaces Implementation For Opportunistic Networks) [BLG15] — JION is a P2P JavaSpace implementation specifically designed for disconnected MANETS where connectivity is unstable.

LogOp [MS03] — LogOp extends basic Linda with coordination primitives for dynamically accessing multiple distributed tuple spaces based on logic expressions.

Scope [MW00] — The concept of *scope* is introduced by Merrick et al. to regulate visibility of tuples. This approach leverages distributed broadcasts for tuple placement and migration.

Lime (Linda In a Mobile Environment) [MPR06] — Lime mobile agents communicate with each other through "transiently shared tuple spaces" whose content is dynamically reconfigured based on the set of co-located agents. That is, Lime deals with both *physical* mobility of *hosts* (changing the actual network topology) and *logical* mobility of *agents* across hosts. Each agent holds a local tuple space. Agents are connected, forming a *group*, when they are co-located in the same host or when they are located in connected hosts. Through *transient sharing*, each agent in the group sees a tuple space given by the merging of all the tuple spaces in the group.

Klaim (Kernel Language for Agent Interaction and Mobility) [DNFP98] — In Klaim, tuples and operations are situated at specific *physical localities* called

sites, whereas programs leverage Linda primitives referring to *logical localities* to basically abstract from actual allocations. A system in Klaim consists of a network of *nodes*, each located at some site, hosting processes. Processes are hence also situated at a site, and issue operations whose target sites depend on an *allocation environment* of the node, which provides a local view of how logical localities map to sites.

3.4 Self-organising coordination

As coordination abstractions of various sorts (e.g., tuple spaces, channels, coordination artefacts [VOR05; Omi+04]) are available in distributed settings, one is directly faced with the problem of dealing with openness (hence, unexpectedness of environment changes, faults, and interactions), large scale (possibly a huge number of agents and coordination abstractions to be managed), and intrinsic adaptiveness (such as the ability to intercept relevant events and react to them to guarantee overall system resilience). This calls for an approach of *self-organising coordination* [VCO09], where coordination abstractions handle "local" interactions only (and typically use stochastic mechanisms to keep the coordination process always "up and running"), such that global and robust patterns of correct coordination behaviour can emerge—achieved by trading off by-design adaptiveness with inherent, automatic forms of adaptiveness.

Coordination models following this approach typically take their inspiration from complex natural systems (from physics through chemistry, all the way to ethology) and attempt to reuse the foundational mechanisms of such systems. A primary source of inspiration for these systems is to be found in biology (social animals, and insects in particular), whose foraging techniques inspire mechanisms to regulate coordination [CVG09; TM04a; Pia+10].

SwarmLinda [TM04a] — SwarmLinda is a tuple-based middleware that brings the collective intelligence displayed by swarms of ants to computational mechanisms aimed at guaranteeing efficient retrieval of tuples. Tuples are handled as forms of pheromones or items that ants (agents) continuously and opportunistically relocate.

Biochemical tuple spaces — Chemical inspiration is used in [VC09; Vir+11] to regulate the "activity level" of tuples, which drives the likelihood of their retrieval as well as their propagation rate. Similarly, ecological inspiration is used in [Vir13] to inject competition, composition, and disposal behaviour in the context of coordination of pervasive computing services.

3.4.1 Field-based coordination

Another important natural source of inspiration comes from physics: a number of physics-inspired self-organising coordination systems rely on the notion of "field" (cf., gravitational field, electromagnetic field), which essentially provides a framework to handle (create, manipulate, combine) global-level, distributed data structures.

A notion of *coordination field* (or co-field) was initially proposed in [MZL03] as a means to support self-organisation patterns of agent movement in complex environments: it was used as an abstraction over the actual environment, spread by both agents and the environment itself, and used by agents (able to locally perceive the value of fields) to properly navigate the environment. Based on this idea, the TOTA (Tuples On The Air) tuple-based middleware [MZ09] was proposed to support field-based coordination for pervasive-computing applications.

TOTA [MZ09] — In TOTA, tuples are associated with *propagation rules* that describe how tuples should be propagated (hop-by-hop) in a network and how the content of tuples should change during propagation. So, these rules determine the *scope* and *transformation* of tuples as they are automatically propagated to neighbour nodes. Moreover, *maintenance* rules are used to define how tuple should *evolve* by reacting to environmental events. In other words, TOTA enables the specification of *dynamic tuple fields*. A derived middleware, *TOTAM (Tuples On The AMbient)* [Boi+14], extends TOTA by evaluating the scope of tuples also *before* transmitting them to neighbour nodes, and including a best-effort leasing model.

Evolving tuples [SJ07] — The *evolving tuples* model is an extension to traditional Linda tuple spaces with the goal of supporting resource discovery in a

pervasive system, relying on ideas similar to those of TOTA. Evolution is firstly embedded in tuples by adding, to each field of the tuple, a name and a formula that specifies the field behaviour over time. Formulas support the if-then-else construct and arithmetic and boolean operators. Secondly, a new operation `evolve`() is introduced in the tuple space, which is responsible for applying formulas to tuples using contextual information.

$\sigma\tau$-Linda [VPB12] — One of the first works connecting field-based coordination with formalisation tools typical of coordination models and languages (e.g., process algebras and transition systems) is the $\sigma\tau$-Linda model, where agents can inject into the space "processes" that spread, collect and decay tuples, ultimately sustaining fields of tuples.

3.5 Final Remarks

This chapter provides a brief and focussed account on research in distributed computing and coordination. Interaction, as an orthogonal dimension to computation, is indeed a crucial aspect of systems, able to give rise to complex behaviour even out of simple processes (as can be observed in many natural systems). Self-organising and field-based coordination, in particular, have largely contributed to spatial and collective adaptive computing approaches (as covered in Chapter 4) and, in turn, to aggregate computing (Chapter 5), the main topic of this thesis.

References

[AG05] Luca Aceto and Andrew D Gordon. "Algebraic process calculi: The first twenty five years and beyond". In: *Process Algebra. http://www. brics. dk/NS/05/3/BRICS-NS-05-3. pdf* (2005).

[Bae05] Jos CM Baeten. "A brief history of process algebra". In: *Theoretical Computer Science* 335.2-3 (2005), pp. 131–146.

[BC91] Antonio Brogi and Paolo Ciancarini. "The Concurrent Language, Shared Prolog". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.1 (1991), pp. 99–123. ISSN: 0164-0925. DOI: 10.1145/114005.102807.

[BG06] Howard Bowman and Rodolfo Gomez. *Concurrency theory: calculi an automata for modelling untimed and timed concurrent systems*. Springer Science & Business Media, 2006.

[BK84] Jan A Bergstra and Jan Willem Klop. "Process algebra for synchronous communication". In: *Information and control* 60.1-3 (1984), pp. 109–137.

[BLG15] Abdulkader Benchi, Pascale Launay, and Frédéric Guidec. "A P2P tuple space implementation for disconnected MANETs". In: *Peer-to-Peer Networking and Applications* 8.1 (2015), pp. 87–102.

[BMS02] Stefania Bandini, Sara Manzoni, and Carla Simone. "Dealing with space in multi–agent systems: a model for situated MAS". In: *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3*. 2002, pp. 1183–1190.

[Boi+14] Elisa Gonzalez Boix, Christophe Scholliers, Wolfgang De Meuter, and Theo D'Hondt. "Programming mobile context-aware applications with TOTAM". In: *Journal of Systems and Software* 92 (2014), pp. 3–19.

[Bus+01] Nadia Busi, Paolo Ciancarini, Roberto Gorrieri, and Gianluigi Zavattaro. "Coordination Models: A Guided Tour". In: *Coordination of Internet Agents: Models, Technologies, and Applications*. Springer, 2001. Chap. 1, pp. 6–24. ISBN: 3-540-41613-7. DOI: 10.1007/978-3-662-04401-8_1.

[BZ07] Mario Bravetti and Gianluigi Zavattaro. "Service oriented computing from a process algebraic perspective". In: *J. Log. Algebr. Program.* 70.1 (2007), pp. 3–14. DOI: 10.1016/j.jlap.2006.05.002. URL: https://doi.org/10.1016/j.jlap.2006.05.002.

[CDK05] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. Pearson Education, 2005.

[Cia96] Paolo Ciancarini. "Coordination models and languages as software integrators". In: *ACM Computing Surveys (CSUR)* 28.2 (1996), pp. 300–302.

[CLZ00] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. "MARS: A Programmable Coordination Architecture for Mobile Agents". In: *IEEE Internet Computing* 4.4 (2000), pp. 26–35. DOI: 10.1109/4236.865084.

[Cor91] Daniel Corkill. "Blackboard Systems". In: *Journal of AI Expert* 9.6 (1991), pp. 40–47.

[CVG09] Matteo Casadei, Mirko Viroli, and Luca Gardelli. "On the collective sort problem for distributed tuple spaces". In: *Science of Computer Programming* 74.9 (2009), pp. 702–722. DOI: 10.1016/j.scico.2008.09.018.

[DNFP98] Rocco De Nicola, Gianluigi Ferrari, and Rosario Pugliese. "KLAIM: A Kernel Language for Agent Interaction and Mobility". In: *IEEE Transaction on Software Engineering (TOSE)* 24.5 (1998), pp. 315–330. ISSN: 0098-5589. DOI: 10.1109/32.685256.

[FHA99] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice: Principles, Patterns and Practices*. The Jini Technology Series. Addison-Wesley Longman, 1999. ISBN: 0201309556.

[Gar02] Vijay K. Garg Ph.D. *Elements of Distributed Computing*. New York, NY, USA: John Wiley & Sons, Inc., 2002. ISBN: 0-471-03600-5.

[Gel85] David Gelernter. "Generative communication in Linda". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7.1 (1985), pp. 80–112. DOI: 10.1145/2363.2433.

[Hoa78] Charles Antony Richard Hoare. "Communicating sequential processes". In: *The origin of concurrent programming*. Springer, 1978, pp. 413–443.

[LDB15] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. "Web service composition: a survey of techniques and tools". In: *ACM Computing Surveys (CSUR)* 48.3 (2015), pp. 1–41.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. San Francisco, USA: Morgan Kaufmann, 1996.

[MC94] Thomas W Malone and Kevin Crowston. "The interdisciplinary study of coordination". In: *ACM Computing Surveys (CSUR)* 26.1 (1994), pp. 87–119.

[Mil09] Robin Milner. *The space and motion of communicating agents*. Cambridge University Press, 2009.

[Mil89] Robin Milner. *Communication and concurrency*. Vol. 84. Prentice hall New York etc., 1989.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The &Pgr;-calculus*. New York, NY, USA: Cambridge University Press, 1999. ISBN: 0-521-65869-1.

[MPR06] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. "LIME: A coordination model and middleware supporting mobility of hosts and agents". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15.3 (2006), pp. 279–328. ISSN: 1049-331X. DOI: 10.1145/1151695.1151698.

[MS03] Ronaldo Menezes and Jim Snyder. "Coordination of Distributed Components Using LogOp". In: *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. Vol. 1. CSREA Press, 2003, pp. 109–114. ISBN: 1-892512-41-6.

[MU00] Naftaly H. Minsky and Victoria Ungureanu. "Law-Governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9.3 (2000), pp. 273–305. ISSN: 1049-331X. DOI: 10.1145/352591.352592.

[MW00] Iain Merrick and Alan Wood. "Scoped coordination in open distributed systems". In: *International Conference on Coordination Languages and Models*. Springer, 2000, pp. 311–316. DOI: 10.1007/3-540-45263-X_21.

[MZ09] Marco Mamei and Franco Zambonelli. "Programming pervasive and mobile computing applications: The TOTA approach". In: *ACM Transactions on Software Engineering Methodologies (TOSEM)* 18.4 (2009), pp. 1–56. ISSN: 1049-331X. DOI: 10.1145/1538942.1538945.

[MZL03] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. "Co-Fields: Towards a Unifying Approach to the Engineering of Swarm Intelligent Systems". In: *Engineering Societies in the Agents World III*. Springer, 2003, pp. 68–81. ISBN: 978-3-540-39173-9. DOI: 10.1007/3-540-39173-8_6.

[OD01] Andrea Omicini and Enrico Denti. "From Tuple Spaces to Tuple Centres". In: *Science of Computer Programming* 41.3 (2001), pp. 277–294. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(01)00011-9.

[Omi+04] Andrea Omicini, Alessandro Ricci, Mirko Viroli, Cristiano Castelfranchi, and Luca Tummolini. "Coordination Artifacts: Environment-Based Coordination for Intelligent Agents". In: *3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IEEE Computer Society, 2004, pp. 286–293. ISBN: 1-58113-864-4.

[PA98] George A Papadopoulos and Farhad Arbab. "Coordination models and languages". In: *Advances in computers*. Vol. 46. Elsevier, 1998, pp. 329–400.

[Pel03] Chris Peltz. "Web services orchestration and choreography". In: *Computer* 36.10 (2003), pp. 46–52.

[Pet66] Carl Adam Petri. "Communication with automata". In: (1966).

[Pia+10] Danilo Pianini, Sascia Virruso, Ronaldo Menezes, Andrea Omicini, and Mirko Viroli. "Self Organization in Coordination Systems Using a WordNet-Based Ontology". In: *4th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2010. DOI: 10.1109/saso.2010.35.

[SJ07] Drew Stovall and Christine Julien. "Resource discovery with evolving tuples". In: *Int. Workshop on Engineering of software services for pervasive environments*. ESSPE. New York, NY, USA: ACM, 2007, pp. 1–10. ISBN: 978-1-59593-798-8. DOI: 10.1145/1294904.1294905.

[TM04a] Robert Tolksdorf and Ronaldo Menezes. "Using Swarm Intelligence in Linda Systems". In: *Engineering Societies in the Agents World IV*. Vol. 3071. Lecture Notes in Computer Science. Springer, 2004, pp. 519–519. ISBN: 978-3-540-22231-6. DOI: 10.1007/978-3-540-25946-6_3.

[TM04b] Robert Tolksdorf and Ronaldo Menezes. "Using Swarm Intelligence in Linda Systems". In: *Engineering Societies in the Agents World IV*. Vol. 3071. Lecture Notes in Computer Science. Springer, 2004, pp. 49–65. ISBN: 3-540-22231-6. DOI: 10.1007/978-3-540-25946-6_3.

[TVS07] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[VC09] Mirko Viroli and Matteo Casadei. "Biochemical Tuple Spaces for Self-organising Coordination". In: *Lecture Notes in Computer Science*. Springer, 2009, pp. 143–162. DOI: 10.1007/978-3-642-02053-7_8.

[VCO09] Mirko Viroli, Matteo Casadei, and Andrea Omicini. "A framework for modelling and implementing self-organising coordination". In: *ACM Symposium on Applied Computing (SAC)*. 2009, pp. 1353–1360. ISBN: 978-1-60558-166-8. DOI: 10.1145/1529282.1529585.

[Vir+11] Mirko Viroli, Matteo Casadei, Sara Montagna, and Franco Zambonelli. "Spatial Coordination of Pervasive Services through Chemical-inspired Tuple Spaces". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 6.2 (2011), 14:1 –14:24. ISSN: 1556-4665. DOI: 10.1145/1968513.1968517.

[Vir13] Mirko Viroli. "On competitive self-composition in pervasive services". In: *Science of Computer Programming* 78.5 (2013), pp. 556 –568. ISSN: 0167-6423. DOI: 10.1016/j.scico.2012.10.002.

[VOR05] Mirko Viroli, Andrea Omicini, and Alessandro Ricci. "Engineering MAS environment with artifacts". In: *2nd International Workshop "Environments for Multi-Agent Systems"(E4MAS 2005)*. AAMAS. 2005, pp. 62–77.

[VPB12] Mirko Viroli, Danilo Pianini, and Jacob Beal. "Linda in Space-Time: An Adaptive Coordination Model for Mobile Ad-Hoc Environments". In: *14th International Conference on Coordination Models and Languages*. 2012, pp. 212–229. DOI: 10.1007/978-3-642-30829-1_15.

[Wyc+98] Peter Wyckoff, Stephen W. McLaughry, Tobin J. Lehman, and Daniel A. Ford. "T Spaces". In: *IBM Journal of Research and Development* 37.3 – Java Techonology (1998), pp. 454–474. ISSN: 0018-8670. DOI: 10.1147/sj.373.0454.

Chapter 4

Spatial and Collective Adaptive Computing: State of the Art

An abstraction is one thing that represents several real things equally well.

Edsger W. Dijkstra

Contents

More or less independently of the problem of finding suitable coordination models for distributed and situated systems, a number of other works investigated the relationship between (local) interaction and global system properties by directly focussing on the *space-time structure* of systems or by directly addressing *ensembles* and *collective behaviour*. Different research communities shared this orientation, often using different terminology, as witnessed by various surveys fo-

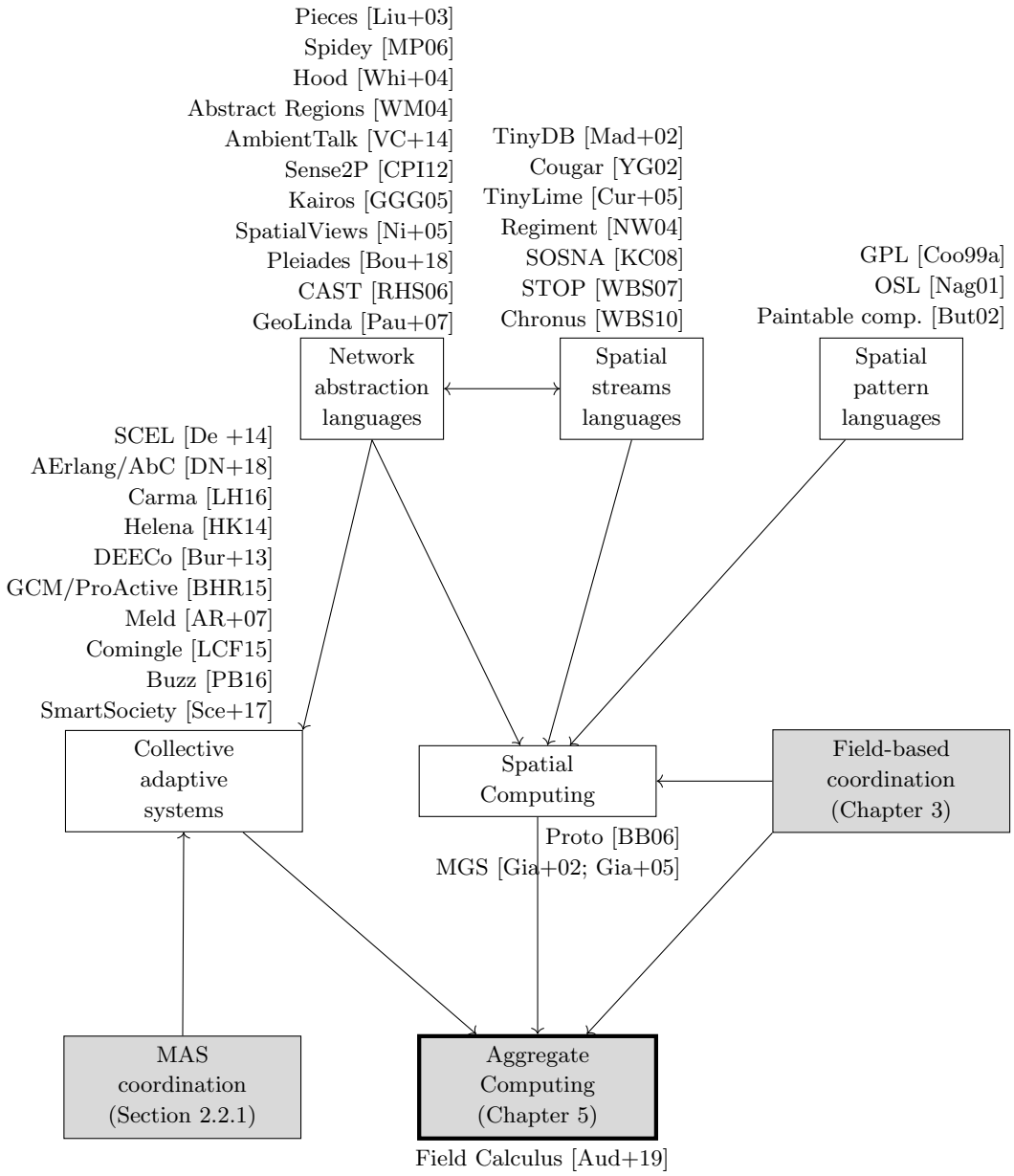


Figure 4.1: Overview of research threads leading from spatial computing and CASs to field calculus and aggregate computing, with some representative bibliographic references. This summary—by no means exhaustive—provides key highlights mainly from the perspective of Chapter 2.

cussing on organisation of aggregates of devices [Bea+13; Vir+19], abstractions for programming WSNs [MP11], or autonomic features [Dob+06; OV11; MT04].

4.1 Spatial Computing Approaches

In [Bea+13], spatial computing languages have found to be used in many domains.

- *Amorphous computing* — This category includes computational approaches focussing on controlling large-scale systems of locally interacting, unreliable devices; these target "amorphous computers", i.e., networks of devices that approximate continuous physical space.
- *Biological modelling and design* — Since biological systems often leverage spatial locality and structure to form or accomplish their functions, approaches in synthetic biology tend to capture such notions for design purposes.
- *Agent-based models* — Since agents are situated entities operating within some environment and organising into structures, multi-agent system modelling and simulation frameworks sometimes provide explicit support for spatial features.
- *Wireless sensor/actuator networks (WSAN)* — These systems are laid on physical environments to gather data, perceive events, and possibly apply local interventions through effectors. As a consequence, languages for WSANs tend to provide mechanisms for controlling topology as well as moving and aggregating data, considering various aspects such as, e.g., energy consumption.
- *Pervasive computing* — Pervasive and ubiquitous systems are inherently integrated into environments and intended to provide contextual services in a seamless way.
- *Swarm and modular robotics* — Swarm robotics leverages locality of interaction to coordinate collective activity in spatial environments, whereas modular robotics approaches address formation and re-configuration of morphologies, often through incremental modifications based on physical contact.

- *Parallel and reconfigurable computing* — In this field, researches focus on hardware architectures and arrangements for efficient computation. Space is relevant because it directly affects communication delays and hence performance.
- *Formal calculi* — Process algebras are used to formally model systems of communicating processes; sometimes, spatial abstractions are provided to deal with issues like communication, situatedness, and mobility.

By neglecting the domains in which spatial abstractions have been used and instead adopting an engineering and programming viewpoint, in the following, an excerpt of relevant spatial computing approaches is provided, by summarising and extending the collection found in the survey [Bea+13]. Works that abstract networks, by using global or spatial abstractions, are covered separately in Section 4.2, whereas those dealing with dynamic collectives are covered in Section 4.3.

4.1.1 Spatial pattern languages

Another class of related approaches falls under the umbrella of languages for expressing geometric constructs and topological patterns. In fact, in several application contexts concerning environment sensing and control, what is key is the physical (geometric, topological) shape that coalitions of mobile agents take, or that certain data items create while diffusing in the environment.

Growing Point Language (GPL) [Coo99b] — In GPL, an *amorphous medium* [Bea05] (essentially defined by an ad-hoc network) can be programmed by a nature-inspired approach of "botanical computing", where computational processes are seen as "growing points" increasingly expanding across neighbours until reaching a fixpoint shape defined by declarative constraints; more specifically, topological structures are formed through attraction and repulsion forces working on growing points (like *tropisms* in plants or insects), driven by simulated chemical signals.

Origami Shape Language (OSL) [Nag08] — OSL is used to achieve similar goals of GPL, though focussing on programming a "computational surface",

intended as a set of small devices working independently of their density in the surface: this language defines geometrical constructs to create basic regions and compose them.

Paintable computing [But02] — A *paintable computer* is essentially an amorphous medium consisting of a large number of tiny *computing particles* that interact with neighbour particles and run asynchronously. Inspired by material self-assembly, this paintable programming model consists of defining a set of process fragments (*pfrags*) that autonomously migrate across the particles and aggregate. Examples of process fragments include gradients, multi-gradients, tessellation processes (creation of Voronoi regions through a multi-gradient from anchor points), diffusion processes, channels, coordinate system formation processes, etc.

4.1.2 General purpose spatial computing languages

General-purpose spatial computing languages address the problem of engineering distributed, concurrent computations by providing mechanisms to manipulate data structures diffused in space and evolving in time. Examples include *StarLisp* [Las+88], the systolic computing programming system *SDEF* [EC89] and, most notably, the following ones.

MGS [Gia+05] — Following a topological computing approach, MGS defines computations over manifolds, the goal of which being to alter the manifold itself as a way to represent input-output transformation.

Computational fields [MZL03; Aud+19] — A *computational field* is a (dynamic) map from a domain of devices to computational objects. The field calculus [Aud+18] is a universal core calculus based on computational fields that underlies the aggregate computing paradigm [Vir+19] (see Chapter 5). Specific programming languages to work with computational fields have been introduced as well, with the Proto [BB06] programming language as common ancestor, Protelis [PVB15] as its Java-oriented DSL version, and ScaFi as presented in this thesis (Chapter 7).

4.2 Network Abstraction and Space-Oriented Macroprogramming Approaches

In this category, there are several approaches, often targetting mobile ad-hoc (MANET) or sensor networks (WSN), that abstract the network of devices itself to simplify various network-wide operations like data collection and event detection. For instance, many of these provide the abstraction of a *region* to capture a collective of nodes through the logical or physical space they occupy; this notion can be defined diversely, e.g.:

- *geometrically*, based on distance metrics;
- *topologically*, based on network links or graph connectivity;
- *logically*, through a matching predicate (cf., *logical neighbourhoods* in *Spidey* [MP06]).

Other approaches use spatio-temporal or macro abstractions to support data gathering and processing in distributed systems.

GeoLinda [Pau+07] — GeoLinda provides geometry-aware distributed tuple spaces where both tuples and reading operations have a spatial extension, or *volume*, called *tuple shape* or *addressing shape*, respectively. Shapes can take various geometric forms (spheres, cones, etc.) and are expressed relatively to a device's location and orientation.

CAST (Coordination Across Space and Time) [RHS06] — CAST is a coordination model aimed at MANETs that leverages mobility to enable operations across space and time. Like other Linda approaches to MANETs, CAST assumes each host has an associated tuple space; moreover, it assumes that hosts move autonomously according to a *motion profile*, advertised through a gossiping protocol. The idea is to leverage knowledge about the motion of other hosts to support operations despite disconnected routes. Operations can refer to remote hosts or spatiotemporal locations: these become *operation requests* which are also associated to a motion profile (scoped on the host network space) and routed to the destination (which may include multiple target hosts) through possibly discon-

nected communication. Write operations (*out*s) are actively performed by sending copies of tuples to hosts entering an area where such tuples are situated. Removal operations (*in*s) are 4-phase: (1) the operation is routed to the targets, (2) an *out* to the originator is issued for each matching tuple in the targets, (3) the originator non-deterministically chooses one tuple, and (4) the host owning the chosen tuple destroys the tuple.

SpatialViews [Ni+05] — This approach works by abstracting a MANET into *spatial views* (i.e., collections of *virtual nodes*) of a configurable space granularity, that can be iterated on to visit nodes and request services. A virtual node is the digital twin of a physical device that has a spatio-temporal location and a set of provided services.

AmbientTalk [VC+14] — It is an ambient-oriented programming language for MANETs based on active objects that provides a classless, prototype-based object model, reified communication traces, ambient acquaintance discovery/failover mechanisms, as well as support for resilience against transient network partitions by automatically buffering asynchronously sent messages.

Hood [Whi+04] — It defines data types to model an agent's neighbourhood and attributes, with operations to read/modify such attributes across neighbours, and a platform optimising execution of such operations by proper caching techniques.

Abstract Regions [WM04] — It provides a *"region-based collective communication interface [...] to hide the details of data dissemination and aggregation within regions"* [WM04]. Supported classes of operators include those for neighbour discovery, enumeration of nodes in a region, data sharing, and data aggregation (or reduction).

Regiment [NW04] — It is a functional reactive, spatiotemporal, macroprogramming language, which models network state as time-varying signals and regions as spatially distributed signals (i.e., computational fields). Regiment provides functional primitives to work with regions. These global operations abstract data acquisition, storage, and communication: it is the job of the compiler to map

these to local operations on the network nodes. Region formation primitives are grouped into two categories: functions for growing regions from "source" points (implemented using spanning trees) and gossip-based functions (based on one-hop broadcasts).

Pieces (Programming and Interaction Environment for Collaborative Embedded Systems) [Liu+03] — Pieces is a *state-centric* programming model where

> *programmers think in terms of dividing the global state of physical phenomena into a hierarchical set of independently updatable pieces with one computational entity (called a* principal*) maintaining each piece.*

That is, a principal is a (possibly mobile) agent that interacts with other principals to update its piece of state. Pieces leverages the notion of *collaboration group*, i.e., a scoped set of principals playing different roles that collaborate to a state update, to abstract communication and resource allocation patterns. Examples of groups include *geographically constrained* groups (a set of nodes located in some geographical region), *n-hop neighbourhood* groups (a set of nodes within n hops from a given anchor node), and *publish/subscribe* groups (a set of consumer and producer nodes on certain topics).

SpaceTime Oriented Programming (STOP) [WBS07] — 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 language is an object-oriented, Ruby DSL enabling on-command and on-demand data collection.

Chronus [WBS10] — Defined as a spatiotemporal macroprogramming language, Chronus is a Ruby DSL, evolved from STOP, for expressing activities of data gathering and event detection in WSNs. A macroprogram in Chronus can specify multiple one-time or periodic queries upon objects representing spacetime regions; instead, event detection is performed against a set of spaces captured through an event specification (predicate); event handlers are functions from the space and time of event occurrences.

Sense2P [CPI12] — It is a *logic macroprogramming system* for WSNs, based on LogicQ [CI08], a system that abstracts sensor networks as relational databases and supports collecting data and spreading logic queries. In Sense2P, programs are expressed in a Prolog-like language: *facts* represent sensor data, and are sent when needed to more powerful nodes (such as base stations) to infer increasingly non-local facts (through *rules*) and answer *queries*.

SOSNA [KC08] — SOSNA is a stream-based, macroprogramming languages for WSANs where programs operate on streams of spatial values. Spatial values are essentially like the regions in Regiment and are called *fields*; the other kind of spatial value is given by *clusters*, which are spatially-limited fields with a singleton node (*cluster head*) holding cluster field data. Execution of SOSNA programs is round-wise and synchronous: a *round* consists of an application-specific number of *steps*; in each step, neighbours exchange a *protocol packet*. Any network operator requires a fixed number of execution steps, and the compiler can statically infer the maximum number of steps of each round. In summary, SOSNA is very similar to Proto [BB06], but requires synchronisation and is limited to WSANs.

Kairos [GGG05] — It is a procedural macroprogramming language for WSNs that assumes loose synchrony (it leverages eventual consistency to keep low overhead) and exposes three main abstractions: named nodes, (one-hop) neighbours, and remote data access.

Pleiades [Bou+18] — It is a topology programming framework leveraging self-organising overlays and assembly-based modularity [Bru+06] to construct and enforce self-stabilising structural invariants in large-scale distributed systems. Shapes

are described through templates specifying positions and neighbours for nodes; configurations of shapes are disseminated in the system and used by nodes for joining shapes; shape formation is regulated through protocols. However, these features are not captured linguistically.

4.3 Collective Adaptive Computing Approaches

In this research area, very related to spatial computing (since systems are often situated and space represents a foundational structure for coordination), it is common to consider large, dynamic groups of devices as first-class abstractions – sometimes called *ensembles*, *collectives*, or *aggregates* – and support interaction between (sub-)groups of devices by abstracting certain details away (e.g., networking, or individual logical connections). With respect to the network abstraction and macroprogramming approaches summarised in Section 4.2, in this section the focus is more on works addressing the specification of dynamic ensembles, that do not take an explicit, spatial space or that are not limited to data gathering and processing.

Helena [HK14] — In Helena ("Handling massively distributed systems with ELaborate ENsemble Architectures"), components can dynamically participate in multiple ensembles and adapt according to different *roles* whose behaviour is given by a process expression.

Distributed Emergent Ensembles of Components (DEECo) [Bur+13] — DEECo is another CAS model where components can only communicate by dynamically binding together through ensembles. A DEECo ensemble is formed according to a *membership condition* and consists of one *coordinator* and multiple *members* interacting by implicit *knowledge exchange*. DEECo has a Java implementation called jDEECo which enables the definition of components and ensembles through Java annotations.

GCM/ProActive [BHR15] — This framework supports the development of large-scale ensembles of adaptable autonomous devices through a hierarchical com-

ponent model where components have a non-functional membrane and "collective interfaces", and a programming model based on active objects.

Service Component Ensemble Language (SCEL) [De +14] — SCEL is a kernel language to specify the behaviour of autonomic components, the logic of ensemble formation, as well interaction through attribute-based communication (which enables implicit selection of a group of recipients).

AErlang/AbC [Alr+15; DN+18] — SCEL is a rich language. A simpler process calculus inspired by SCEL is *AbC (Attribute-based Communication)* [Alr+15]. AbC has been implemented for the Erlang programming language through the AErlang library [DN+18].

CARMA [LH16] — CARMA (acronym for Collective Adaptive Resource-sharing Markovian Agents) is a stochastic process algebra and language that models collective of components that may dynamically aggregate into ensembles. It uses attribute-based communication (uni- or multi-cast) to coordinate large ensembles of devices via local broadcast operations.

Meld [AR+07] — Meld is a logic programming language for modular robotics. It abstracts low-level coordination in robot ensembles by taking a global perspective to programming. Production rules are used to generate new facts from existing ones to possibly enable other rules (forward chaining); facts that are invalidated will be *eventually* deleted; aggregate rules are used to collapse multiple facts into one.

Comingle [LCF15] — Inspired by Meld, Comingle is a distributed, logic programming framework for systems of mobile devices. In this model, devices are identified through a location and contribute to the system through *located facts*; the set of all located facts is called a *rewriting state*; *rules* operate on rewriting states. The rewriting semantics is global: it operates on a distributed data structure of located facts (contributed by the locations participating in the ensemble).

Buzz [PB16] — Buzz is a swarm-oriented programming system. In Buzz, a *swarm* consists of a set of robots equipped with the Buzz virtual machine and running the same Buzz script in a step-by-step fashion. In each step, a robot (i) collects sensor readings and incoming messages; (ii) executes a portion of the Buzz script; (iii) sends output messages; and (iv) applies actuators on actuator values hold in the state. Robots can share information through *virtual stigmergy* [PLBB16] or by querying neighbours. The language comprises both single-robot and swarm-based primitives.

SmartSociety platform — In [Sce+17], a programming model of SmartSociety for hybrid collaborative adaptive systems is proposed in which the designer specifies an environment where collectives—i.e., persistent or transient teams of peers (humans and machines)—are involved in collective tasks.

In the above approaches, the ensemble abstraction is dynamic—in order to cope with change—and hence provides a way to adapt the coordination logic.

4.4 Final Remarks

As the complexity of computer-based artificial systems increases – because of more (heterogeneous) components, more interconnections, and more abstraction layers – so is the need for techniques capturing collective behaviour, adaptivity, and global aspects of systems. The approaches surveyed in this chapter contribute to this quest by leveraging space-time and macro abstractions. However, they generally target only specific domains (cf. query languages for WSNs, or swarm programming languages), provide only basic mechanisms (e.g., ensemble membership, collective communication interfaces), lack the ability to compose simple collective behaviours together to build more complex ones, or are too abstract (providing only a core calculus that does not easily map to effective implementations). Among the approaches consider, the field calculus seems particularly suitable as a general-purpose, "scalable" model for expressing how local activity turns into collective adaptive behaviour. A more in-depth presentation of this framework and the corresponding programming paradigm, called *aggregate computing*, is provided

in Chapter 5. The rest of the thesis builds on aggregate computing research to provide contributions – in terms of model extensions, tools, and patterns – for the development complex collective adaptive (eco-)systems.

References

[Alr+15] Yehia Abd Alrahman, Rocco De Nicola, Michele Loreti, Francesco Tiezzi, and Roberto Vigo. "A calculus for attribute-based communication". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 2015, pp. 1840–1845.

[AR+07] Michael P Ashley-Rollman, Seth Copen Goldstein, Peter Lee, Todd C Mowry, and Padmanabhan Pillai. "Meld: A Declarative Approach to Programming Ensembles". In: *International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2007, pp. 2794–2800. DOI: 10.1109/IROS.2007.4399480.

[Aud+18] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. "Space-time universality of field calculus". In: *International Conference on Coordination Languages and Models*. Springer. 2018, pp. 1–20.

[Aud+19] Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. "A Higher-Order Calculus of Computational Fields". In: *ACM Transactions on Computational Logic* 20.1 (2019), pp. 1–55. DOI: 10.1145/3285956.

[BB06] Jacob Beal and Jonathan Bachrach. "Infrastructure for Engineered Emergence in Sensor/Actuator Networks". In: *IEEE Intelligent Systems* 21 (2 2006), pp. 10–19. DOI: 10.1109/MIS.2006.29.

[Bea05] J. Beal. "Amorphous medium language". In: *Large-Scale Multi-Agent Systems Workshop (LSMAS)*. Available at http://jakebeal.com/. 2005, pp. 1–7.

[Bea+13] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. "Organizing the Aggregate: Languages for Spatial Computing". In: *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. A longer version available at: http://arxiv.org/abs/1202.5509. IGI Global, 2013. Chap. 16, pp. 436–501. ISBN: 978-1-4666-2092-6. DOI: 10.4018/978-1-4666-2092-6.ch016.

[BHR15] Françoise Baude, Ludovic Henrio, and Cristian Ruz. "Programming distributed and adaptable autonomous components—the GCM/ProActive framework". In: *Software: Practice and Experience* 45.9 (2015), pp. 1189–1227. DOI: 10.1002/spe.2270.

[Bou+18] Simon Bouget, Yérom-David Bromberg, Adrien Luxey, and François Taiani. "Pleiades: Distributed structural invariants at scale". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2018, pp. 542–553.

[Bru+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. "The fractal component model and its support in java". In: *Software: Practice and Experience* 36.11-12 (2006), pp. 1257–1284.

[Bur+13] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. "DEECO: an ensemble-based component system". In: *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*. ACM. 2013, pp. 81–90. DOI: 10.1145/2465449.2465462.

[But02] William Butera. "Programming a Paintable Computer". PhD thesis. Cambridge, USA: MIT, 2002.

[CI08] Supasate Choochaisri and Chalermek Intanagonwiwat. "A system for using wireless sensor networks as globally deductive databases". In: *2008 IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*. IEEE. 2008, pp. 649–654.

[Coo99a] Daniel Coore. "Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer". PhD thesis. Cambridge, MA, USA: MIT, 1999.

[Coo99b] Daniel Coore. "Botanical computing: a developmental approach to generating interconnect topologies on an amorphous computer". PhD thesis. Massachusetts Institute of Technology, 1999.

[CPI12] Supasate Choochaisri, Nuttanart Pornprasitsakul, and Chalermek Intanagonwiwat. "Logic macroprogramming for wireless sensor networks". In: *International Journal of Distributed Sensor Networks* 8.4 (2012), p. 171738.

[Cur+05] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. "Mobile data collection in sensor networks: The TinyLime middleware". In: *Elsevier Pervasive and Mobile Computing Journal* 4 (2005), pp. 446–469. DOI: 10.1016/j.pmcj.2005.08.003.

[De +14] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. "A Formal Approach to Autonomic Systems Programming: The SCEL Language". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 9.2 (2014), 7:1–7:29. DOI: 10.1145/2619998.

[DN+18] Rocco De Nicola, Tan Duong, Omar Inverso, and Catia Trubiani. "AErlang: empowering Erlang with attribute-based communication". In: *Science of Computer Programming* 168 (2018), pp. 71–93.

[Dob+06] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. "A survey of autonomic communications". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 1.2 (2006), pp. 223–259. DOI: 10.1145/1186778.1186782.

[EC89] Bradley R. Engstrom and Peter R. Cappello. "The SDEF programming system". In: *Journal of Parallel and Distributed Computing* 7.2 (1989), pp. 201–231. DOI: 10.1016/0743-7315(89)90018-X.

[GGG05] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. "Macroprogramming wireless sensor networks using Kairos". In: *International Conference on Distributed Computing in Sensor Systems*. Springer. 2005, pp. 126–140.

[Gia+02] Jean-Louis Giavitto, Christophe Godin, Olivier Michel, and Przemyslaw Prusinkiewicz. *Computational models for integrative and developmental biology*. Tech. rep. 72-2002. Univerite d'Evry, LaMI, 2002.

[Gia+05] Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. "Computations in Space and Space in Computations". In: *Unconventional Programming Paradigms*. Vol. 3566. Lecture Notes in Computer Science. Berlin: Springer, 2005, pp. 137–152. ISBN: 978-3-540-27884-9. DOI: 10.1007/11527800_11.

[HK14] Rolf Hennicker and Annabelle Klarl. "Foundations for ensemble modeling—the Helena approach". In: *Specification, Algebra, and Software*. Springer, 2014, pp. 359–381. DOI: 10.1007/978-3-642-54624-2_18.

[KC08] Marcin Karpiński and Vinny Cahill. "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. 2008, pp. 49–55.

[Las+88] C. Lasser, J.P. Massar, J. Miney, and L. Dayton. *Starlisp Reference Manual*. Thinking Machines Corporation, 1988.

[LCF15] Edmund Soon Lee Lam, Iliano Cervesato, and Nabeeha Fatima. "Comingle: Distributed logic programming for decentralized mobile ensembles". In: *International Conference on Coordination Languages and Models*. Springer. 2015, pp. 51–66.

[LH16] Michele Loreti and Jane Hillston. "Modelling and Analysis of Collective Adaptive Systems with CARMA and its Tools". In: *Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems - 16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2016, Bertinoro, Italy, June 20-24, 2016, Advanced Lectures*. Ed. by Marco Bernardo, Rocco De Nicola, and Jane Hillston. Vol. 9700. Lecture Notes in Computer Science. Springer, 2016, pp. 83–119. ISBN: 978-3-319-34095-1. DOI: 10.1007/978-3-319-34096-8_4. URL: https://doi.org/10.1007/978-3-319-34096-8_4.

[Liu+03] Jie Liu, Maurice Chu, J Reich, and F Zhao. "State-centric programming for sensor-actuator network systems". In: *IEEE Pervasive Computing* 2.4 (2003), pp. 50–62.

[Mad+02] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. "TAG: A Tiny AGgregation Service for Ad-hoc Sensor Networks". In: *SIGOPS Operating System Review* 36.SI (2002), pp. 131–146. ISSN: 0163-5980. DOI: 10.1145/844128.844142.

[MP06] Luca Mottola and Gian Pietro Picco. "Logical neighborhoods: A programming abstraction for wireless sensor networks". In: *International Conference on Distributed Computing in Sensor Systems*. Springer. 2006, pp. 150–168.

[MP11] Luca Mottola and Gian Pietro Picco. "Programming wireless sensor networks: Fundamental concepts and state of the art". In: *ACM Computing Surveys (CSUR)* 43.3 (2011), p. 19.

[MT04] Ronaldo Menezes and Robert Tolksdorf. "Adaptiveness in Linda-Based Coordination Models". In: *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering*. Vol. 2977. LNAI. Springer, 2004, pp. 212–232. ISBN: 3-540-21201-9. DOI: 10.1007/b95863.

[MZL03] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. "Co-Fields: Towards a Unifying Approach to the Engineering of Swarm Intelligent Systems". In: *Engineering Societies in the Agents World III*. Springer, 2003, pp. 68–81. ISBN: 978-3-540-39173-9. DOI: 10.1007/3-540-39173-8_6.

[Nag01] Radhika Nagpal. "Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics". PhD thesis. Cambridge, MA, USA: MIT, 2001.

[Nag08] Radhika Nagpal. "Programmable pattern-formation and scale-independence". In: *Unifying themes in complex systems IV*. Springer, 2008, pp. 275–282.

[Ni+05] Yang Ni, Ulrich Kremer, Adrian Stere, and Liviu Iftode. "Programming ad-hoc networks of mobile and resource-constrained devices". In: *ACM SIGPLAN Notices* 40.6 (2005), pp. 249–260.

[NW04] Ryan Newton and Matt Welsh. "Region Streams: Functional Macroprogramming for Sensor Networks". In: *Workshop on Data Management for Sensor Networks*. Toronto, Canada, 2004, pp. 78–87. DOI: 10.1145/1052199.1052213.

[OV11] Andrea Omicini and Mirko Viroli. "Coordination Models and Languages: From Parallel Computing To Self-Organisation". In: *The Knowledge Engineering Review* 26.1 (2011), pp. 53–59. ISSN: 0269-8889. DOI: 10.1017/S026988891000041X.

[Pau+07] Julien Pauty, Paul Couderc, Michel Banatre, and Yolande Berbers. "Geo-linda: a geometry aware distributed tuple space". In: *21st International Conference on Advanced Information Networking and Applications (AINA'07)*. IEEE. 2007, pp. 370–377.

[PB16] Carlo Pinciroli and Giovanni Beltrame. "Buzz: An extensible programming language for heterogeneous swarm robotics". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2016, pp. 3794–3800.

[PLBB16] Carlo Pinciroli, Adam Lee-Brown, and Giovanni Beltrame. "A tuple space for data sharing in robot swarms". In: *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*. 2016, pp. 287–294.

[PVB15] Danilo Pianini, Mirko Viroli, and Jacob Beal. "Protelis: practical aggregate programming". In: *Symposium on Applied Computing*. ACM. 2015, pp. 1846–1853. DOI: 10.1145/2695664.2695913.

[RHS06] Gruia-Catalin Roman, Radu Handorean, and Rohan Sen. "Tuple space coordination across space and time". In: *International Conference on Coordination Languages and Models*. Springer. 2006, pp. 266–280.

[Sce+17] Ognjen Scekic, Tommaso Schiavinotto, Svetoslav Videnov, Michael Rovatsos, Hong-Linh Truong, Daniele Miorandi, and Schahram Dustdar. "A Programming Model for Hybrid Collaborative Adaptive Systems". In: *IEEE Transactions on Emerging Topics in Computing* (2017).

[VC+14] Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. "AmbientTalk: programming responsive mobile peer-to-peer applications with actors". In: *Computer Languages, Systems & Structures* 40.3-4 (2014), pp. 112–136.

[Vir+19] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. "From distributed coordination to field calculus and aggregate computing". In: *Journal of Logical and Algebraic Methods in Programming* (2019), p. 100486. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2019.100486.

[WBS07] Hiroshi Wada, Pruet Boonma, and Junichi Suzuki. "A spacetime oriented macroprogramming paradigm for push-pull hybrid sensor networking". In: *2007 16th International Conference on Computer Communications and Networks*. IEEE. 2007, pp. 868–875.

[WBS10] Hiroshi Wadaa, Pruet Boonmab, and Junichi Suzukic. "Chronus: A spatiotemporal macroprogramming language for autonomic wireless sensor networks". In: *Autonomic Network Management Principles: From Concepts to Applications* (2010), p. 167.

[Whi+04] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. "Hood: a neighborhood abstraction for sensor networks". In: *2nd International Conference on Mobile systems, applications, and services*. Boston, MA, USA: ACM, 2004. DOI: 10.1145/990064.990079.

[WM04] Matt Welsh and Geoffrey Mainland. "Programming Sensor Networks Using Abstract Regions." In: *NSDI*. Vol. 4. 2004, pp. 3–3.

[YG02] Yong Yao and Johannes Gehrke. "The Cougar Approach to In-Network Query Processing in Sensor Networks". In: *ACM Sigmod record* 31.3 (2002), pp. 9–18. DOI: 10.1145/601858.601861.

Chapter 5

Aggregate Computing

Strength lies in union.

Aesop · *The Old Man and his Sons*

Contents

Aggregate programming is a paradigm for developing collective adaptive systems by a global perspective that emerged from the amorphous programming language Proto [BB06] and previous work covered in Chapter 4. In this chapter, we review the mathematical core of aggregate computing, i.e., the *field calculus* (Section 5.1), together with its most relevant formalisations and properties. Then, we cover additional ingredients needed to evolve a basic field calculus implementation to a full aggregate computing support (Section 5.2), and accordingly present

the Protelis programming language [PVB15] and a set of aggregate functions representing a foundational library for collective adaptive behaviour.

5.1 Field Calculus

In this section, we describe a formal framework based on the notion of *computational field* (see Section 4.1.2), i.e., a "collective data structure" that maps elements of a domain (e.g., space-time localities, or devices) to computational values. By distributing devices and making them compute over time, the computational field abstraction takes the form of a distributed, dynamic data structure capturing the result of collective behaviour. Therefore, the idea for compositional specification of increasingly complex collective behaviour lies in the ability to functionally manipulate such collective results. The operational idea, instead, involves bridging the local with the global: this is achieved through neighbourhood-based interaction, whereby local results affect increasingly non-local results and vice versa. Finally, adaptivity stems from an *execution model* whereby devices repeatedly (i) sample their local context, (ii) locally interpret the field specification against that context, and (iii) propagate up-to-date results (called *exports*) to their neighbours. In other words, field programs generally describe continuous, collectively-coherent manipulations of "context fields" to obtain "output fields" with useful global-level properties—by controlled emergence.

5.1.1 Basic calculus

The field calculus (FC) was introduced in [VDB13] as a minimal core calculus meant to capture the key ingredients of languages that make use of computational fields:[1] functions over fields, functional composition with fields, evolution of fields over time, construction of fields of values from neighbours, and restriction of a field computation to a sub-region of the network.

The field calculus is based on the idea of specifying the aggregate system behaviour of a network of devices, where a dynamic neighbouring relation (which is application-dependent and represents physical or logical proximity) is used to

[1] This is similar to how λ-calculus [Chu32] captures the essence of functional computation and FJ [IPW01] the essence of class-based object-oriented programming.

P	::=	$\overline{\texttt{F}}$ e	program
F	::=	def d($\overline{\texttt{x}}$) {e}	function declaration
e	::=	x | v | f($\overline{\texttt{e}}$) | if(e){e}{e} | nbr{e} | rep(e){(x)=>{e}}	expression
f	::=	d | b	function name
v	::=	ℓ | ϕ	value
ℓ	::=	c($\overline{\ell}$)	local value
ϕ	::=	$\overline{\delta} \mapsto \overline{\ell}$	neighbouring field value

Figure 5.1: Abstract syntax of the field calculus, as adapted from [Vir+18]

indicate the devices with which one can directly communicate[2]—e.g., in a sensor network, those within the range of a broadcast communication. One such specification is structured as a functional composition of operators that manipulate (evolve, combine, restrict) computational fields.

A key feature of the approach is that a specification can be interpreted either locally or globally. Locally, it can be seen as describing a computation on an individual device, iteratively executed in asynchronous *computation rounds* comprising reception of messages from neighbours, perception of contextual information through sensors, storing local state of computation, computing the local value of fields, and spreading messages to neighbours. Globally, a field calculus expression e specifies a mapping (i.e., the computational field) associating each computation round of each device to the value that e assumes at that space-time event. This duality intrinsically supports the reconciliation between the local behaviour of each device and the emerging global behaviour of the whole network of devices [DVB16; VDB13], as proved by the computational adequacy and abstraction properties in [Aud+19b], which relate operational and denotational semantics.

computation rounds

Figure 5.1 gives an abstract syntax for field calculus, as presented in recent works [Vir+18]. In this syntax, the overbar notation $\overline{\texttt{e}}$ indicates a sequence of elements (i.e., $\overline{\texttt{e}}$ stands for $\texttt{e}_1, \texttt{e}_2, \ldots, \texttt{e}_n$), and multiple overbars are expanded together (e.g., $\overline{\delta} \mapsto \overline{\ell}$ stands for $\delta_1 \mapsto \ell_1, \delta_2 \mapsto \ell_2, \ldots, \delta_n \mapsto \ell_n$ which is a map associating local values to device identifiers). There are four keywords in this

[2]A device with no neighbours, e.g., would be one isolated (temporarily or permanently) from the rest of the system.

syntax: `def` for function definition; `if` for (the field-based variation of) branching expression; and `rep` and `nbr` for the two peculiar constructs of field calculus, respectively responsible for evolution of state over time and for sharing information between neighbours.

A field calculus program `P` consists of a sequence of function declarations $\overline{\texttt{F}}$ followed by the main expression `e`, defining global (and also local) behaviour of the aggregate system. An expression `e` can be:

- A *variable* `x`, e.g., a function parameter.
- A *value* `v`, which can be of the following two kinds:
  - a *local value* ℓ, defined via data constructor `c` and arguments $\overline{\ell}$, such as a Boolean, number, string, pair, tuple, etc;
  - a *neighbouring (field) value* ϕ representing a collection of values from nearby devices, in the form of a function that associates, for each device, the set of neighbour devices δ (including the device itself) to local values ℓ, e.g., a map of neighbours to the distances to those neighbours.
- A function call $\texttt{f}(\overline{\texttt{e}})$ to either a *user-declared function* `d` (declared with the `def` keyword) or a *built-in function* `b`, such as a mathematical or logical operator, a data structure operation, or a function returning the value of a sensor.
- A *branching expression* $\texttt{if}(\texttt{e}_1)\{\texttt{e}_2\}\{\texttt{e}_3\}$, used to split a computation into isolated sub-regions where (and when) $\texttt{e}_1$ evaluates to `True` or `False`: the result is computation of $\texttt{e}_2$ in the former area, and $\texttt{e}_3$ in the latter.
- The `nbr{e}` construct, which creates a neighbouring value mapping neighbours to their latest available result of evaluating `e`. In particular, each device δ:
  1. shares its value of `e` with its neighbours, and
  2. evaluates the expression into a neighbouring value ϕ, where ϕ is a function that maps each neighbour δ' of δ to the latest evaluation of `e` that has been shared from δ'.

  For instance, `nbr{temperature()}` (where `temperature` is a built-in sensor estimating local temperature) would produce a neighbouring value ϕ associ-

```
// distance from source region with nbrRange metric
def distanceTo(source) {
  rep (Infinity) { (dist) =>
     mux ( source, 0, minHood(nbr{dist} + nbrRange()) )
  }
}
// distance from source region, avoiding obstacle region
def distanceToWithObs(source, obstacle) {
  if (obstacle) { Infinity }{ distanceTo(source) }
}
// main expression
distanceToWithObs(deviceId() == 0, senseObs())
```

Figure 5.2: Example field calculus code

ating to each neighbour the temperature measured by that neighbour. Note that in an `if`, sharing is restricted to occur between devices within the same subspace of the branch (since devices in a different subspace do not execute the same `nbr{e}` constructs).

- The `rep`(e_1){`(x)=>{}`}e_2`}` construct, which models state evolution over time. This construct retrieves the value `v` computed for the whole `rep` expression in the last evaluation round (the value produced by evaluating the expression e_1 is used at the first evaluation round) and updates it with the value produced by evaluating the expression obtained from e_2 by replacing the occurrences of `x` by `v`.

Within this collection of operations, the `nbr` and `rep` constructs are special, handling message exchanges respectively between devices and within rounds of a single device. These constructs are assumed to be backed by a data gathering mechanism accomplished through a process called *alignment* [Aud+16], which ensures appropriate message matching, i.e., that no two different instances of a `nbr` expression can inadvertently "swap" their respective messages, nor can two different instances of a `rep` expression "swap" their state memory. This has the notable consequence that the two branches of an `if` statement in field calculus are executed in isolation: a device computing the "then" branch cannot communicate with the "else" branch of a neighbour, and vice versa.

alignment

Example 5.1 (Distance Avoiding Obstacles)**.** *Consider Figure 5.2. Function* `distanceTo` *takes as argument a field of Booleans* `source`*, associating true to* source nodes*, and produces as result a field of reals, mapping each device to its minimum distance to a source node, as computed by relaxation of the triangle inequality; namely: repetitively, and starting from infinity (construct* `rep`*) everywhere, the distance on any node gets updated to* 0 *on source nodes (function* `mux(c,t,e)` *is a purely functional multiplexer which chooses* `t` *if* `c` *is* `true`*, or* `e` *otherwise), and elsewhere to the minimum (built-in* `minHood`*) of neighbours' distance (construct* `nbr`*) added with* `nbrRange`*, a sensor for estimated distances. Function* `distanceToWithObs` *takes an additional argument, a field of Booleans* `obstacle`*, associating true to* obstacle nodes*; it partitions the space of devices: on obstacle nodes it gives the field of infinity values, elsewhere it uses computation of* `distanceTo`*. Because of alignment, the set of neighbours considered for* `distanceTo` *automatically discards nodes that evaluate the other branch of* `if`*, effectively making computation of distances circumvent obstacles. Finally, the main expression calls* `distanceToWithObs` *to compute distances from the node with* `deviceId` *equal to* `0`*, circumventing the devices where* `senseObs` *gives true.*

Example 5.2 (Monitor)**.** *Consider the following field calculus expression.*

```
if ( fail() ) { rep (0) {(x) => x-1} } { sumHood(nbr{1}) }
```

This expression represents a simple monitor, for which higher values indicate a good situation, while lower (negative) values signal problematic situations. In devices where `fail` *is true, the number of consecutive rounds of failure is counted with negative numbers by the* `rep` *expression. Non-failing devices instead compute* `sumHood(nbr{1})` *(isolated from failing devices) which* (i) *builds a neighbouring field* ϕ *mapping each non-failing neighbour to* 1*;* (ii) *sums every value in the range of* ϕ *(except that for the current device) with built-in* `sumHood`*, obtaining the (non-negative) total number of non-failing neighbours.*

5.1.2 Operational semantics, typing and basic properties

small-step operational semantics

The distinguished interaction model of this approach has been first formalised in [VDB13] (see also [DVB16]) by means of a *small-step operational semantics*

modelling single device computation (which is ultimately responsible for the whole network execution). The main technical novelty in this formalisation is that device state and message content are represented in a unified way as an annotated evaluation tree θ. Field construction, propagation, and restriction are then supported by local evaluation "against" the collection Θ of evaluation trees received from neighbours. The alignment mechanism to ensure appropriate message matching is then implemented by operations navigating these trees, and discarding them whenever different branches are taken (to prevent unwanted communication between `nbr` constructs in different branches of an `if` expression).

big-step operational semantics

Recent work models single device computation by a *big-step operational semantics* [Vir+18], expressed by the judgement $\delta; \Theta; \sigma \vdash \mathtt{e}_{\mathtt{main}} \Downarrow \theta$, to be read "expression $\mathtt{e}_{\mathtt{main}}$ evaluates to θ on device δ with respect to environment Θ and sensor state σ". The overall network evolution is then formalised by a small-step operational semantics as a transition system $N \xrightarrow{act} N$ on network configurations N, in which actions *act* can either be environment changes or single device computations (in turn modelled by the big-step semantics).

field calculus type system

The work in [DVB16] presents a type system, used to intercept ill-formed field-calculus programs, which builds on the *Hindley-Milner type system* [DM82] for ML-like functional languages, as a set of syntax-directed type inference rules. Being syntax-directed, the rules straightforwardly specify a variant of the Hindley-Milner type inference algorithm [DM82]. Namely, an algorithm that, given a field calculus expression and type assumptions for its free variables, either fails (if the expression cannot be typed under the given type assumptions) or returns its principal type, i.e., a type such that all the types that can be assigned to an expression by the type inference rules can be obtained from the principal type by substituting type variables with types. This type system is proved to guarantee the following two valuable properties for field calculus:

- *Domain alignment:* On each device, the domain of every neighbouring value arising during the reduction of a well-typed expression consists of the identifiers of the aligned neighbours and of the identifier of the device itself. In other words, information sharing is scoped to precisely implement the aggregate abstraction.
- *Type soundness:* The reduction of a well-typed expression does not get stuck.

Example 5.3 (Typing). *Consider the Examples 5.1 and 5.2. The type system assigns the following types to the involved built-in functions, user-defined functions, and main expressions.*

```
// minHood, sumHood : (bool) -> num
// nbrRange : () -> field(num)
def distanceTo(source) ... // (bool) -> num
def distanceToWithObs(source, obstacle) ... // (bool, bool) -> num
distanceToWithObs(deviceId() == 0, senseObs()) // num
if ( fail() ) { rep (0) {(x) => x-1} } { sumHood(nbr{1}) } // num
```

5.1.3 Behavioural properties

The field calculus is designed as a general-purpose language for spatially distributed computations.

self-stabilisation

Thus, regularity properties have been isolated and studied for subsets of the core language. Among them, the established notion of *self-stabilisation* to correct states for distributed systems [Dol00; LLM17; LLM15] plays a central role. This notion, defined in terms of properties of the transition system $N \xrightarrow{act} N$ of network evolution (cf. Section 5.1.2), ensures that both *(i)* the evaluation of a program on an eventually constant input converges to a limit value in each device in finite time; *(ii)* this limit only depends on the input values (i.e., sensor values and neighbouring links), and not on the transitory input values that may have happened before that. When applied in a dynamically evolving system, a self-stabilising algorithm guarantees that whenever the input changes, the output reacts accordingly without spurious influences from past values.

self-stabilising fragments

In [DV15] (an extended version of [VD14]), a first self-stabilising fragment is isolated through a *spreading* operator, which minimises neighbour values as they are monotonically updated by a *diffusion* function. This pattern can be composed arbitrarily with local operations, but no explicit `rep` and `nbr` expressions are allowed: nonetheless, several building blocks can be expressed inside this fragment, such as classic distance estimation and broadcast (specific instances of operator G in Figure 5.4).

More self-stabilising programs and existing "building block" implementations are covered by the larger self-stabilising fragment introduced in [Vir+18] (an ex-

tended version of [Vir+15]). This fragment restricts the usage of `rep` statements to three specific patterns: converging, acyclic, and minimising `rep`. They roughly correspond to the three main building blocks proposed, G, C and T: G is a generalisation of distance estimation, which spreads a spanning tree from a source region based on a given metric, and use it to compute values outward; C conversely collects values inward a spanning tree (typically produced by G) aggregating them "en route" so as to summarise a final result into a target node; and finally T is a local operator to temporally evolve a value until reaching a fixpoint—see Figure 5.4). Furthermore, a notion of *equivalence* and *substitutability* for self-stabilising programs is examined: on the one hand, this notion allows for practical optimisation of distributed programs by substitution of routines with equivalent but better-performing alternatives; on the other hand, this equivalence relation naturally induces a *limit* viewpoint for self-stabilising programs, complementing and integrating the two general (local and global) viewpoints by abstracting away the transitory characteristics and isolating the input-output mapping corresponding to the distributed algorithm. These viewpoints effectively constitute different semantic interpretations of the same program: operational semantics (local viewpoint), denotational semantics (global viewpoint), and eventual behaviour (limit viewpoint).

equivalence and substitution

A fourth "continuous" viewpoint is considered in [Bea+17]: as the density of computing devices in a given area increases, assuming that each device takes inputs from a single continuous function on a space-time manifold, the output values may converge towards a limit continuous output. Programs with this property are called *consistent*, and have a "continuous" semantic interpretation as a transformation of continuous functions on space-time manifolds. Taking inspiration from self-stabilisation, this notion is relaxed for *eventually consistent* programs, which are only required to continuously converge to a limit except for a transitory initial period, *provided* that the inputs are constant (except for a transitory initial period). Eventual consistency can then be proved for all programs expressible in the *GPI (gradient-following path integral) calculus*, which is a restriction of the field calculus where the only coordination mechanism allowed is the GPI operator, a generalised variant of the distance estimation building block.

space-time consistency

GPI calculus

Finally, a recent thread of work [Aud+18] has begun considering the transient

real-time guarantees

behaviour of field calculus programs, by providing real-time guarantees on program performance. In these results, a bounded amount of error with respect to ideal values is proved to hold after a predictable set-up (or reconfiguration) time.

Up to this point, hence, validation of behavioural properties is mostly addressed "by construction", namely, proving properties on simple building blocks or restricting the calculus to fragments. It is a future work to consider the applicability of techniques such as the formal basis in [LLM17], or model-based analysis such as [Bak+11].

5.1.4 Language extension: the higher-order field calculus

higher-order field calculus (HFC)

The *higher-order field calculus (HFC)* [Aud+19b; Dam+15] is an extension of the field calculus with first-class functions. Its primary goal is to allow programmers to handle functions just like any other value, so that code can be dynamically injected, moved, and executed in network (sub)domains. Namely, in HFC:

- Functions can take functions as arguments and return a function as result (higher-order functions). This is key to define highly reusable building block functions, which can then be fully parametrised with various functional strategies.
- Functions can be created "on the fly" (anonymous functions). Among other applications, such functions can be passed into a system from the external environment, as a field of functions considered as input coming from a sensor modelling addition of new code into a device while the system is operating.
- Functions can be moved between devices (via the `nbr` construct) and the function to be executed can be remembered and changed over time (via the `rep` construct), which allows one to express complex patterns of code deployment across space and time.
- A field of functions (possibly created on the fly and then shared by movement to all devices) can be used as an "aggregate function" operating over a whole spatial domain.

In considering fields of function values, HFC takes an approach in which making a function call acts as a branch, with each function in the range of the field applied only on the subspace of devices that hold that function. When the field of functions

is constant, this implicit branch reduces to be precisely equivalent to a standard function call. This means that we can view ordinary evaluation of a function name (or anonymous function) as equivalent to creating a function-valued field with a constant value, then making a function call applying that field to its argument fields. This elegant transformation is one of the key insights of HFC, enabling first-class functions to be implemented with relatively minimal complexity.

In [Dam+15] the operational semantics of HFC is formalised, for computation within a single device, by a big-step operational semantics where each expression evaluates to an ordered tree of values tracking the results of all evaluated sub-expressions. Moreover, [Dam+15] also presents a formalisation of network evolution, by a transition system on network configurations—transitions can either be firings of a device or network configuration changes, while network configurations model environmental conditions (i.e., network topology and inputs of sensors on each device) and the overall status of devices in the network at a given time. In the extension of this work in [Aud+19b] the formalisation of HFC is carried on by providing a denotational semantics, which is proved to correspond to the operational semantics through *computational adequacy and abstraction* results. Furthermore, a refined type system is presented that is able to guarantee *domain alignment*, i.e., that the domain of any expression of field type equals the set of neighbours that computed the same expression.

5.2 From Field Calculus to Aggregate Computing

In this section, we discuss the current state of the art in practical aggregate computing. We begin by discussing the construction of implementations of field calculus as supported by the domain specific language Protelis (Section 5.2.1). We then discuss the layered abstractions of aggregate programming built upon these foundations, from resilient operators to pragmatic libraries (Section 5.2.2). Note that as far as current implementations are concerned, field calculus is supported in its higher-order version, hence in the following we sometimes generally refer to field calculus even if higher-order capabilities are concerned.

5.2.1 Protelis: a DSL for field calculus

The concrete usage of field calculus in application development is dependent on the availability of practical languages, which provide an interpreter or compiler, as well as handling runtime aspects such as communication, interfacing with the operating system, and integration with existing software. Protelis [PVB15] provides one such implementation, including: (i) a concrete syntax; (ii) an interpreter and a virtual machine; (iii) a device interface abstraction and API; and (iv) a communication interface abstraction and API.

In Protelis, the parser translates a Protelis source code file into a valid representation of HFC semantics. This translated program, along with an execution context, is fed to a virtual machine that executes the Protelis interpreter at regular intervals. The execution context API defines the interface towards the operating system, including (with ancillary APIs) an abstraction of the device's capabilities and communication system. This architecture has been demonstrated to make the language easy to port across diverse contexts, both simulated (Alchemist[PMV13] and NASA World Wind [Bel+07]) and real-world [CBP15].

The entire Protelis infrastructure is developed in Java and hosted on the Java Virtual Machine (JVM). The motivation behind this choice is twofold: first, the JVM is highly portable, being available on a variety of architectures and operating systems; second, the Java world is rich in libraries that can be directly used within Protelis, with little or no need for writing new libraries for common tasks.

The model-to-model translation between the Protelis syntax and the HFC interpreter is implemented using the Xtext framework [Bet16]. Along with the parser machinery, this framework is also able to generate most of the code required for implementing Eclipse plug-ins: one such plug-in is available for Protelis, assisting the developer through code highlighting, completion suggestions, and early error detection.

The language syntax is designed with the goal of lowering the learning curve for the majority of developers, and as such it is inspired by languages of the C-family (C, C++, Java, C#, ...), with some details borrowed from Python. Code can be organised in modules (or namespaces) whose name must reflect the directory structure and the file name. Modules can contain functions and a main script. The code snippet in Figure 5.3 offers a sampler of both the ordinary and

```
import protelis:coord:spreading // Import other modules
import java.lang.Math.sqrt // Import static Java methods
def privateFun(my, params) {
  my + params // Infix operators, duck typing
}
public def availableOutside() { // externally visible
  privateFun(1, 2); // Function call
  let aFun = privateFun; // Variable definition, function ref
  aFun.apply("a", "str"); // String literals, application
  let tup = [NaN, pi, e]; // Tuple literals, built-in numbers
  // lambda expressions, closures, method invocation:
  let inc3 = v -> {privateFun(v, tup.size())}
}
// MAIN SCRIPT
let myid = self.getDeviceUID(); // Access to device info
if (myid < 1000) { // Domain separation
  rep (x <- self.nextRandomDouble()) {// Stateful computation
    // Java static method call
    mux (sqrt(x) < 0.5) { // mux executes both branches
      // Library call, field gathering and reduction
      minHood(nbr(env.has("source")))
    } else { Infinity }
  } < 10
} else { // Mandatory else: every expression returns a value
  false // Booleans
}
```

Figure 5.3: Example Protelis code showcasing a sampler of language features.

field-calculus-specific features of Protelis, including importing libraries and static methods, using functions as higher-order values in `let` constructs and by `apply`, tuple and string literals, lambdas, built-ins (e.g., `minHood`, and `mux`), and the field calculus constructs `rep` and `nbr`.

Function definitions are prefixed by the `def` keyword, and they are visible by default only in the local module. In order for other modules to access them, the keyword `public` must be explicitly specified. Other modules can be imported, as well as Java static methods. Types are not specified explicitly: in fact, Protelis is duck-typed—namely, type-checked at run-time through reflection mechanisms. The language offers literals for commonly used numeric values, tuples, and strings. Instance methods can be invoked on any expression with the same "dot" syn-

tax used in Java. Higher order support includes a compact syntax for lambda expressions, closures, function references, functions as parameters, and function application. Lastly, context properties, including device capabilities, are accessible through the `self` keyword. Environment variables can be accessed via the short syntax `env`.

Another relevant asset of Protelis is its library `protelis-lang` [Fra+17], streamlining the implementation of a number of algorithms found in the distributed systems literature. Among others, it includes several implementations of self-stabilising building block functions [BV14; Vir+18], such as `distanceTo` to estimate distances, `broadcast` to send alerts, `summarize` to perform distributed sensing, and so on. Notably, the library also includes meta-machinery for "aligning" aggregate computing programs along arbitrary keys, separating and mixing domains in a finer way than the `if` construct allows. These constructs, based on the `alignedMap` primitive of Protelis, enable highly dynamic meta-algorithms to be written, that open up new possibilities such as `multiInstance` [Fra+17], or allow for increased resilience and adaptation as in the case of `timeReplicated` [PBV16].

5.2.2 Aggregate Programming

Building upon these theoretical and pragmatic foundations, aggregate programming [BPV15] elaborates a layered architecture that aims to dramatically simplify the design, creation, and maintenance of complex distributed systems. This approach is motivated by three key observations about engineering complex coordination patterns:

- composition of modules and subsystems must be simple and transparent;
- different subsystems need different coordination mechanisms for different regions and times;
- mechanisms for robust coordination should be hidden by abstractions, such that programmers are not required to interact with the details of their implementation.

Field calculus (along with its language incarnations) provides mechanisms for the first two, but is too general to guarantee resilience and too mathematical and

succinct in its syntax for direct programming to be simple: some methodology is needed to properly scale with complexity.

Aggregate programming thus proposes two additional abstraction layers, as illustrated in Figure 5.4, for hiding the complexity of distributed coordination in complex networked environments. First, the "resilient coordination operators" layer plays a crucial role both in hiding the complexity and in supporting efficient engineering of distributed coordination systems. First proposed in [BV14], it is inspired by the approach of combinatory logic [CF58], the catalogue of self-organisation primitives in [FM+13], and work on self-stabilising fragments of the field calculus [DV15; Vir+18; VD14]. Notably, three key operators within this self-stabilising fragment cover a broad range of distributed coordination patterns: operator `G` is a highly general information spreading and "outward computation" operation; `C` is its inverse, a general information collection operation; and `T` implements bounded state evolution and short-term memory. *G-C-T*

Above the resilience layer, aggregate programming libraries [Fra+17; Vir+15] capture common patterns of usage and more specialised and efficient variants of resilient operators to provide a more user-friendly interface for programming. This definition of well-organised layers of abstractions with predictable compositional semantics thus aims to foster *(i) reusability*, through generic components; *(ii) productivity*, through application-specific components; *(iii) declarativity*, through high-level functionality and patterns; *(iv) flexibility*, through low-level and fine-grained functions; and *(v) efficiency*, through multiple components with coherent substitution semantics [Vir+18; Vir+15].

Within these two layers, development has progressed from an initial model built only around the spreading of information to a growing system of composable operators and variants. The first of these operator/variant families to be developed centred around the problems of spreading information, since interaction in aggregate computing is often structured in terms of information flowing through collectives of devices. A major problem thus lies in regulating such spreading, in order to take into account context variation, and in rapidly adapting the spreading structure in reaction to changes in the environment and in the system topology. Here, the gradient (i.e., the field of minimum distances from source nodes) in its generalised form in the `G` operator is what captures, in a distributed way, a no-

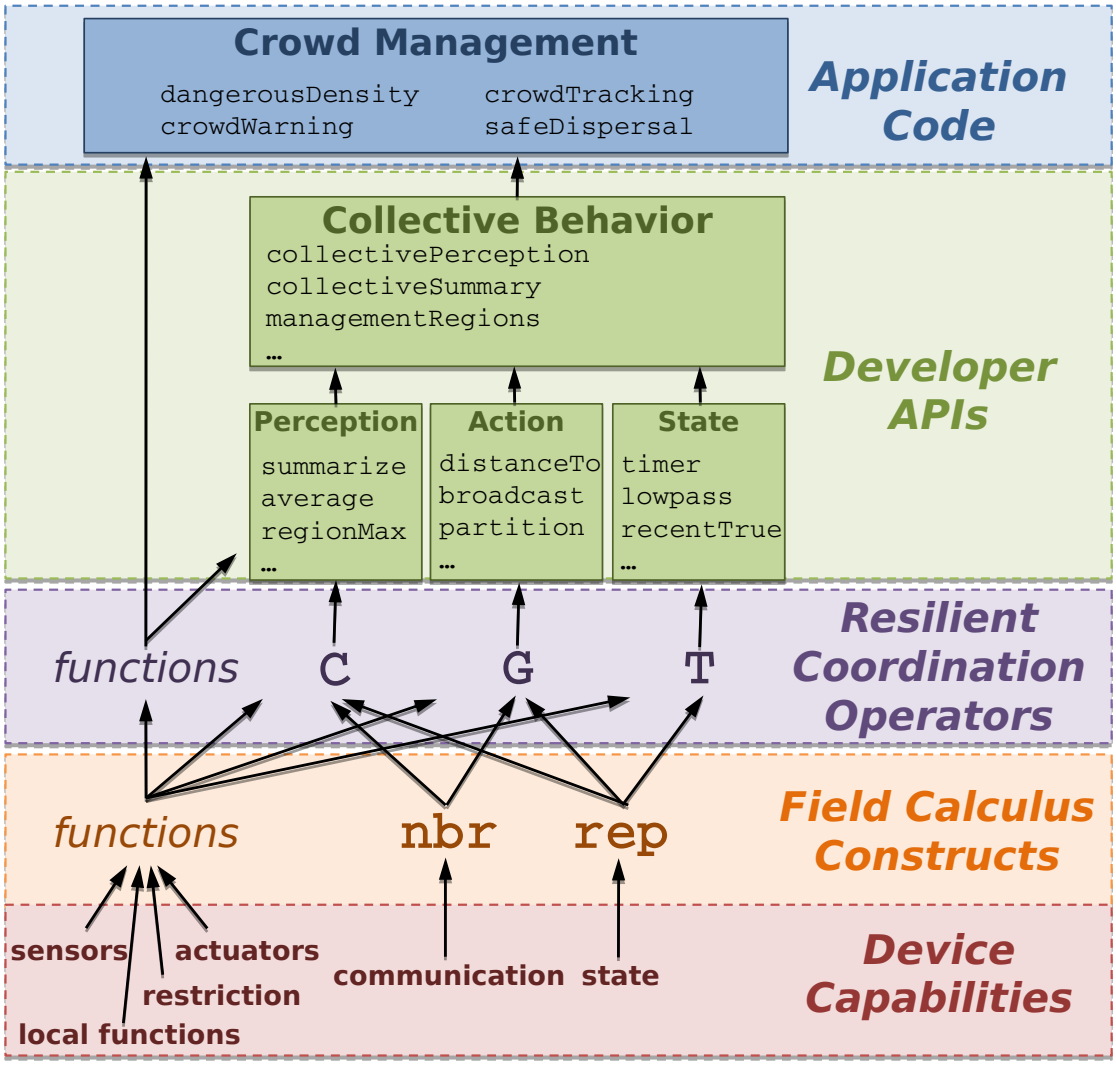


Figure 5.4: Aggregate programming abstraction layers. The software and hardware capabilities of particular devices are used to implement aggregate-level field calculus constructs. These constructs are used to implement a limited set of building-block coordination operations with provable resilience properties, which are then wrapped and combined together to produce a user-friendly API for developing situated IoT (Internet-of-Things) systems. Figure adapted from [BPV15].

tion of "contextual distance" instrumental for calculating information diffusion, and forms the basis for key interaction patterns, such as outward/inward bounded broadcasts and dynamic group formation, as well as higher-level components built upon these.

The widespread adoption of gradient structures in algorithms stresses the importance of fast self-healing gradients [Bea+08], which are able to quickly recover good distance estimates after disruptive perturbations, and more "dependable" gradient algorithms in which stability is favoured by enacting a smoother self-healing behaviour [Bea09]. Several other alternative gradient algorithms have also been developed, addressing two main issues. Firstly, the recovery speed after an input discontinuity, which has first been bounded to $O(diameter)$ time by the CRF (constraint and restoring force) gradient algorithm [Bea+08], further improved to optimal for algorithms with a single-path communication pattern by the BIS (bounded information speed) gradient algorithm [ADV18], and refined to optimality for algorithms with a multi-path communication pattern by the
SVD (stale values detection) gradient algorithm [Aud+17]. Secondly, the smoothness and resilience to noise in inputs, first addressed by the FLEX (flexible) gradient algorithm [Bea09] and then refined and combined with improved recovery speed by the ULT (ultimate) gradient algorithm [Aud+17].

To empower the aggregate programming tool-chain, other building blocks have been proposed and refined in addition to gradients: consensus algorithms [Bea16], centrality measures [ADV17], leader election and partitioning [BV14], and most notably, *collection* [Vir+18; Vir+15]. The collection building block `C` progressively aggregates and summarises values spread throughout a network into a single value, e.g., their sum or other meaningful statistics. Based itself on distance estimation through gradients, a general *single-path* collection algorithm has been proposed in [BV14] granting self-stabilisation to a correct value, then *multi-path* collection has been developed for improved resiliency in sum estimations [Vir+18], and finally refined to *weighted multi-path* collection [AB17] and its parametric extension [Aud+19a], which is able to maintain acceptable whole-network sums and maxima even in highly volatile environments. A different approach to collection has also proved to be effective for *minimum/maximum* estimates: overlapping replicas of non-self-stabilising *gossip* algorithms [PBV16] (with an appropriately tuned

interval of replication), thus combining the resiliency of these algorithms with self-stabilisation requirements.

In sum, the current state of aggregate computing features pragmatic implementations of field calculus supporting an expanding library of resilient building blocks with various trade-offs in their dynamical behaviour, and which can be used as the basis for implementation of a wide variety of distributed applications.

5.3 Final Remarks

Aggregate computing is a macro-paradigm enabling functional composition of collective adaptive behaviour, based on the field calculus. At the current state, there are four main issues:

- implementations do not seamlessly integrate with mainstream programming environments (though Protelis made progress through JVM interoperability);
- by a programming model perspective, there is a lack of mechanisms able to effectively capture concurrent, dynamic field computations;
- there is no aggregate computing middleware to support the development and deployment of distributed aggregate applications; and
- there is still little experience and guidance regarding the design of aggregate systems.

This work aims to fill such a gap, as covered in Part II (Chapters 7 to 10).

References

[AB17] Giorgio Audrito and Sergio Bergamini. "Resilient Blocks for Summarising Distributed Data". In: *Proceedings of the First Workshop on Architectures, Languages and Paradigms for IoT, ALP4IoT@iFM 2017*. Ed. by Danilo Pianini and Guido Salvaneschi. Vol. 264. EPTCS. 2017, pp. 23–26. DOI: 10.4204/EPTCS.264.3.

[ADV17] Giorgio Audrito, Ferruccio Damiani, and Mirko Viroli. "Aggregate Graph Statistics". In: *Proceedings of the First Workshop on Architectures, Languages and Paradigms for IoT, ALP4IoT@iFM 2017*. Ed. by Danilo Pianini and Guido Salvaneschi. Vol. 264. EPTCS. 2017, pp. 18–22. DOI: 10.4204/EPTCS.264.2.

[ADV18] Giorgio Audrito, Ferruccio Damiani, and Mirko Viroli. "Optimal single-path information propagation in gradient-based algorithms". In: *Sci. Comput. Program.* 166 (2018), pp. 146–166. DOI: 10.1016/j.scico.2018.06.002.

[Aud+16] Giorgio Audrito, Ferruccio Damiani, Mirko Viroli, and Roberto Casadei. "Run-Time Management of Computation Domains in Field Calculus". In: *Foundations and Applications of Self* Systems, IEEE International Workshops on.* IEEE. 2016, pp. 192–197.

[Aud+17] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. "Compositional Blocks for Optimal Self-Healing Gradients". In: *Self-Adaptive and Self-Organising Systems (SASO), IEEE International Conference on.* IEEE. 2017.

[Aud+18] Giorgio Audrito, Ferruccio Damiani, Mirko Viroli, and Enrico Bini. "Distributed Real-Time Shortest-Paths Computations with the Field Calculus". In: *2018 IEEE Real-Time Systems Symposium (RTSS).* IEEE, 2018, pp. 23–34. DOI: 10.1109/rtss.2018.00013.

[Aud+19a] Giorgio Audrito, Sergio Bergamini, Ferruccio Damiani, and Mirko Viroli. "Effective Collective Summarisation of Distributed Data in Mobile Multi-Agent Systems". In: *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19.* 2019, pp. 1618–1626. URL: http://dl.acm.org/citation.cfm?id=3331882.

[Aud+19b] Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. "A Higher-Order Calculus of Computational Fields". In: *ACM Transactions on Computational Logic* 20.1 (2019), pp. 1–55. DOI: 10.1145/3285956.

[Bak+11] Rena Bakhshi, Lucia Cloth, Wan Fokkink, and Boudewijn R. Haverkort. "Mean-field framework for performance evaluation of push–pull gossip protocols". In: *Performance Evaluation* 68.2 (2011). Advances in Quantitative Evaluation of Systems, pp. 157 –179. ISSN: 0166-5316. DOI: 10.1016/j.peva.2010.08.025.

[BB06] Jacob Beal and Jonathan Bachrach. "Infrastructure for Engineered Emergence in Sensor/Actuator Networks". In: *IEEE Intelligent Systems* 21 (2 2006), pp. 10–19. DOI: 10.1109/MIS.2006.29.

[Bea+08] Jacob Beal, Jonathan Bachrach, Dan Vickery, and Mark Tobenkin. "Fast self-healing gradients". In: *Proceedings of the 2008 ACM symposium on Applied computing.* ACM. 2008, pp. 1969–1975.

[Bea09] Jacob Beal. "Flexible self-healing gradients". In: *Proceedings of the 2009 ACM symposium on Applied Computing.* ACM. 2009, pp. 1197–1201.

[Bea16] Jacob Beal. "Trading accuracy for speed in approximate consensus". In: *The Knowledge Engineering Review* 31.4 (2016), pp. 325–342.

[Bea+17] Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. "Self-adaptation to Device Distribution in the Internet of Things". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 12.3 (2017), p. 12. DOI: 10.1145/3105758.

[Bel+07] David G. Bell, Frank Kuehnel, Chris Maxwell, Randy Kim, Kushyar Kasraie, Tom Gaskins, Patrick Hogan, and Joe Coughlan. "NASA World Wind: Open-source GIS for Mission Operations". In: *Aerospace Conference*. IEEE, 2007. DOI: 10.1109/aero.2007.352954.

[Bet16] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend, 2E*. Packt Publishing, 2016. ISBN: 1786464969, 9781786464965.

[BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. "Aggregate Programming for the Internet of Things". In: *IEEE Computer* 48.9 (2015), pp. 22–30. DOI: 10.1109/MC.2015.261.

[BV14] Jacob Beal and Mirko Viroli. "Building Blocks for Aggregate Programming of Self-Organising Applications". In: *8th International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*. 2014, pp. 8–13. DOI: 10.1109/SASOW.2014.6.

[CBP15] Shane S Clark, Jacob Beal, and Partha Pal. "Distributed recovery for enterprise services". In: *9th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE. 2015, pp. 111–120. DOI: 10.1109/SASO.2015.19.

[CF58] H.B. Curry and R. Feys. *Combinatory logic*. North-Holland, 1958.

[Chu32] Alonzo Church. "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics*. Second Series 33.2 (1932), pp. 346–366. DOI: 10.2307/1968337.

[Dam+15] Ferruccio Damiani, Mirko Viroli, Danilo Pianini, and Jacob Beal. "Code Mobility Meets Self-organisation: A Higher-Order Calculus of Computational Fields". English. In: *Formal Techniques for Distributed Objects, Components, and Systems*. Vol. 9039. Lecture Notes in Computer Science. Springer, 2015, pp. 113–128. ISBN: 978-3-319-19194-2. DOI: 10.1007/978-3-319-19195-9_8.

[DM82] Luis Damas and Robin Milner. "Principal Type-schemes for Functional Programs". In: *Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: ACM, 1982, pp. 207–212. ISBN: 0-89791-065-6. DOI: 10.1145/582153.582176.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[DV15] Ferruccio Damiani and Mirko Viroli. "Type-based Self-stabilisation for Computational Fields". In: *Logical Methods in Computer Science* 11.4 (2015).

[DVB16] Ferruccio Damiani, Mirko Viroli, and Jacob Beal. "A type-sound calculus of computational fields". In: *Science of Computer Programming* 117 (2016), pp. 17 –44. ISSN: 0167-6423. DOI: 10.1016/j.scico.2015.11.005.

[FM+13] Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, Sara Montagna, Mirko Viroli, and Josep Lluis Arcos. "Description and composition of bio-inspired design patterns: a complete overview". In: *Natural Computing* 12.1 (2013), pp. 43–67. ISSN: 1572-9796. DOI: 10.1007/s11047-012-9324-y.

[Fra+17] Matteo Francia, Danilo Pianini, Jacob Beal, and Mirko Viroli. "Towards a Foundational API for Resilient Distributed Systems Design". In: *International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. IEEE, 2017. DOI: 10.1109/fas-w.2017.116.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: A Minimal Core Calculus for Java and GJ". In: *ACM Transactions on Programming Languages and Systems* 23.3 (2001), pp. 396–450.

[LLM15] Alberto Lluch-Lafuente, Michele Loreti, and Ugo Montanari. "A Fixpoint-Based Calculus for Graph-Shaped Computational Fields". In: *17th International Conference on Coordination Models and Languages (COORDINATION)*. 2015, pp. 101–116. DOI: 10.1007/978-3-319-19282-6_7.

[LLM17] Alberto Lluch-Lafuente, Michele Loreti, and Ugo Montanari. "Asynchronous Distributed Execution Of Fixpoint-Based Computational Fields". In: *Logical Methods in Computer Science* 13.1 (2017). DOI: 10.23638/LMCS-13(1:13)2017. URL: https://doi.org/10.23638/LMCS-13(1:13)2017.

[PBV16] Danilo Pianini, Jacob Beal, and Mirko Viroli. "Improving Gossip Dynamics Through Overlapping Replicates". In: *Proceedings of the 18th International Conference on Coordination Models and Languages*. Vol. 9686. Lecture Notes in Computer Science. Springer, 2016, pp. 192–207. DOI: 10.1007/978-3-319-39519-7_12.

[PMV13] Danilo Pianini, Sara Montagna, and Mirko Viroli. "Chemical-oriented Simulation of Computational Systems with Alchemist". In: *Journal of Simulation* (2013). ISSN: 1747-7778. DOI: 10.1057/jos.2012.27.

[PVB15] Danilo Pianini, Mirko Viroli, and Jacob Beal. "Protelis: practical aggregate programming". In: *Symposium on Applied Computing*. ACM. 2015, pp. 1846–1853. DOI: 10.1145/2695664.2695913.

[VD14] Mirko Viroli and Ferruccio Damiani. "A Calculus of Self-stabilising Computational Fields". In: *16th International Conference on Coordination Models and Languages (COORDINATION)*. Vol. 8459. Lecture Notes in Computer Science. Springer, 2014, pp. 163–178. DOI: 10.1007/978-3-662-43376-8_11.

[VDB13] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. "A Calculus of Computational Fields". In: *Advances in Service-Oriented and Cloud Computing*. Vol. 393. Communications in Computer and Information Science. Springer, 2013, pp. 114–128. ISBN: 978-3-642-45363-2. DOI: 10.1007/978-3-642-45364-9_11.

[Vir+15] Mirko Viroli, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. "Efficient Engineering of Complex Self-Organising Systems by Self-Stabilising Fields". In: *9th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. 2015, pp. 81–90. DOI: 10.1109/SASO.2015.16.

[Vir+18] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. "Engineering Resilient Collective Adaptive Systems by Self-Stabilisation". In: *ACM Transaction on Modelling and Computer Simulation* 28.2 (2018), 16:1–16:28. ISSN: 1049-3301. DOI: 10.1145/3177774.

Chapter 6

Complex Infrastructures and Deployments

We shape our buildings, thereafter they shape us.

Winston Churchill

Contents

The evolution of the science and practice of computers and networks has always been characterised by an increase in terms of *abstraction*—with the sporadic revaluation and re-adjustment of important details that were abstracted away. Think

of the role of operating systems, the "write once, run everywhere" motto of Java, the ISO/OSI networking layers, the conceptual/logical/physical levels in database management systems, and cluster/cloud management software. Think also of the problems with the *remote procedure call* paradigm [TR88], or the issues of cloud computing for latency-sensitive applications.

A related, key concept in modern software-based system engineering is *DevOps*. As a contraction of *Development* and *Operations*, this term refers to the shortening of the gap between these two engineering aspects in order to improve reliability and agility in deployment. DevOps is strongly based on automation and programmability of operational processes.

In this chapter, the current trends in computing and application deployment are highlighted, together with the perspectives most related to collective and pervasive computing. These themes are prominent in this thesis, since (i) aggregate computing, through its declarative model, fosters operational flexibility in terms of both targets and adaptivity of execution plans; (ii) distinction between logical and physical systems fosters separation of concerns (cf. Chapter 9); and (iii) aggregate techniques are shown to be useful to program the "infrastructure" – modelled as a physical computational space – and to enable collective adaptive behaviour on various levels and kinds of infrastructural support (cf. Chapter 10).

6.1 Fundamentals

6.1.1 Virtualisation

virtualisation

Virtualisation is the practice of creating virtual counterparts for physical resources. This is related but different from *emulation* and *simulation*, whose goal is to accurately reproduce or approximate the behaviour of the emu-/simu-lated entity, respectively.

Virtual machines Things that could be useful to virtualise include computers: their virtual counterparts are called *virtual machines (VMs)*. Virtual machines can be run as *guest machines* on physical, *host machines* through a *hypervisor*. Hypervisors can be distinguished between [PG74]:

hypervisor

- *Type-1, native, or bare-metal hypervisors*, which run directly on the host's hardware — Examples include Xen or Microsoft Hyper-V.
- *Type-2, or hosted hypervisors*, which run on the host operating system — Examples include VMware, VirtualBox, and QEMU.

full virtualisation

In *full virtualisation*, hardware is replicated (*hardware virtualisation*) so that the guest OS can run unaware of the abstraction involved. *Hardware-assisted* or *accelerated virtualisation* leverages certain hardware capabilities to improve the efficiency of full virtualisation. *Paravirtualisation*, by contrast, is a technique where the guest OS is modified to perform (hyper-)calls on the hypervisor.

A tool for creating, running, managing VMs in a reproducible and portable way is *Vagrant*: it uses `Vagrantfile`s to declaratively describe VM configuration and provisioning, possibly reusing other VMs (*boxes*), and supports multiple *providers* (e.g., VirtualBox, Hyper-V, etc.).

Containers In *application* or *process virtualisation*, an application is run in an isolated fashion on the underlying OS. This is supported by *OS-level* or *lightweight virtualisation*, whereby kernel mechanisms are used to enable user-space isolation. Also known as *containerisation*, this technique helps to run isolated applications, called *containers*, without incurring in the overhead of virtual machines, where hardware has to be virtualised and a full guest OS run.

containers and containerisation

A prominent toolkit for working with containers is *Docker*. It provides, among others, a notion of *image* (a template for creating containers), easy construction of images via `Dockerfile`, registries for storing images, support for networking between containers, volumes for data sharing, and tools, e.g., for defining and running multi-container applications (Docker Compose) and managing a cluster of Docker hosts (Docker Swarm).

6.1.2 Management platforms

Platforms exist to manage heterogeneous systems composed of multiple servers, networks, storage, and application resources.

Mesos [Ign16] — Mesos is a *cluster management system* that abstracts physical resources in order to enable multiple frameworks to share resources in a fine-grained fashion (cf. static partitioning), improving cluster utilisation. The architecture of Mesos consists of a *master node* (among a set of candidate masters coordinating through ZooKeeper) interacting with a set of *agents*, i.e., cluster nodes that advertise the resources they offer to the master and are available for running tasks. One or more *frameworks* register with the master to be offered resources; the master uses an *allocation module* to decide how many resources to offer frameworks; the frameworks can *reject* resource offers until they receive satisfactory ones; it is the framework's *scheduler* component that takes decisions about whether accepting or rejecting offers (e.g., based on workload). When a framework accepts a resource offer, it sends a description of the tasks to run to the master, which in turn launches the tasks on the agents (these are executed by a *framework executor process*).

6.2 Cloud Computing

cloud computing

cloud characteristics

Cloud computing [EPM13] is an ICT environment for remote, elastic provisioning of resources and services. Cloud environments are characterised by on-demand usage (i.e., quick self-provisioning), elasticity (i.e., the ability to automatically scale in/out), multi-tenancy (resource pooling and virtualisation are used to serve multiple tenants in a more-or-less isolated way), pay-per-use (resource usage is measured for monitoring and billing), resilience (i.e., reliability and availability through replication of resources), and remote, ubiquitous access (e.g., through global connectivity). Enabling technologies include networks, virtualisation, service technology and the Web, and data centre technology (for consolidating and managing resources). Cloud services are commonly classified according to three main *delivery models*:

cloud delivery models

- *Infrastructure-as-a-Service (IaaS)* — the provider makes available to subscribers networking, storage, and server resources.
- *Platform-as-a-Service (PaaS)* — in addition to IaaS, the provider also makes available OS and runtime platform to subscribers, which only need to provide applications on top;

- *Software-as-a-Service (SaaS)* — the cloud provides a full stack from infrastructure to applications.

Since more things can be provided "as a service", cloud computing is often referred to as Everything-as-a-Service (XaaS). For instance, modern delivery models include *Container-as-a-Service (CaaS)* (where the cloud enables deployment of containerised microservices), and *Function-as-a-Service (FaaS)* (the cloud enables deployment and activation, through configurable triggers, of functions—this is also known as a "serverless architecture", since "servers are abstracted away"), where pricing is based on actual usage of resource (at a fine granularity).

Example: Google Cloud Platform (GCP) GCP is Google's cloud computing platform. Users can interact with GCP via four main modalities: (i) the Cloud Console (a Web interface), (ii) the REST API, (iii) the command line tool `gcloud`, or (iv) programmatically, through client libraries (with bindings for multiple programming languages) that use the REST API. The set of tools for building software that uses GCP is known as the *Google Cloud Software Development Kit (SDK)*. An *account* can group resources into isolated containers known as *projects*. Cloud resources are consolidated into so-called *data centers (DC)*. The locations of DCs can be important for latency and isolation. A *zone* denotes a single physical facility: it is the smallest unit in which a resource can exist. If two resources are localised in the same zone, it means that zone-level failures can affect them at the same time. A *region* denotes a collection of zones; it has the approximate extension of a city. Depending on how services are deployed across zones and regions, increasing *levels of isolation* can be considered: *zonal* (services in a single zone), *regional* (services in a single region), *multi-regional* (services in different regions), and *global* (services spread around the world, for maximal isolation). GCP provides a plethora of services spanning the categories of computing, storage, networking, security, etc. Examples include: *Compute Engine*, for on-demand deployment of virtual machines; *Kubernetes Engine*, for Kubernetes clusters; *App Engine*, a PaaS for managed applications; *Cloud Functions*, for serverless applications; *Cloud SQL*, for managed relational databases (MySQL or PostgreSQL); *Cloud Datastore*, for managed document storage; *Cloud Storage*, for managed object-into-bucket storage. The cost of use of services depends on the specific service considered; however,

typical cost components include how much data is stored, how much data is transferred (ingress and egress), how much computation and memory is used per time, how many requests or invocations are performed, etc.

Cloud management: OpenStack [SAE12] — OpenStack is an open-source platform that supports the management of heterogeneous, physical and virtual resources through a common API. In other words, it is sort of "cloud OS" that provides a management layer for data centres, enabling the creation of private and public clouds. OpenStack consists of various components that focus on specific cloud features: *OpenStack Compute (Nova)* manages infrastructure and provides virtual machines; *OpenStack Networking (Neutron)* manages all aspects related to networking; *OpenStack Orchestration (Heat)* provides an orchestration platform for template-based cloud applications; etc.

6.3 Beyond Cloud Computing: Edge and Fog Computing

Recently, the Cloud Computing paradigm [MG+11] has become mainstream, reliably providing elastic, virtually unlimited, on-demand resources (i.e., services, processing power, storage) to users through Internet connectivity and large data centres sparse around the globe. Together with the progress in data centre management, early efforts were directed at exploiting one such model for disparate scenarios, leading to fields such as *Mobile Cloud Computing (MCC)* [FLR13]. In MCC, the goal is to give a support to mobile devices for *offloading* data and computations to the cloud [MB17]. However, such an approach requires communicating with distant data centres, leading to high latency, energy, and bandwidth consumption, and additional load on mobile networks. In order to solve these issues, the idea is to bring cloud-like functionality closer to where resources are needed. Several concepts (with similar or overlapping meanings) emerged in this direction, including *cyber foraging* [SKK12], *cloudlets* [Sat+09], as well as *ad-hoc* [Yaq+16], *peer-to-peer* [BMT12], and *mobile edge-clouds* [FLR16]. These efforts have been somewhat subsumed by *fog* [Bon+12; VRM14] and *(mobile) edge*

mobile cloud computing

computing [Hu+15; Shi+16] paradigms, whose goal is indeed to support computation at the edge of the network.

fog computing, edge computing

Fog and Mobile Edge Computing (FMEC) leverage dense geographical deployments of resource providers and infrastructural elements, as well as the corresponding proximity to users, to both enable new scenarios and improve the efficiency of the system itself and the Quality of Service (QoS) of applications running atop. Indeed, FMEC is not to be intended as a replacement for traditional Cloud Computing but rather as a *complementary paradigm*, providing options to system designers when, e.g.: the cloud is (temporarily or permanently) not available; the cloud is available but undesirable or incompatible with cost, latency or other non-functional requirements; or when the kind of services to be provided operate inherently at the edge (cf., mobile crowdsensing [GYL11]). Accordingly, FMEC represents a facilitator for Internet of Things (IoT) and smart city applications [DB16], where humans, intelligence, and control are largely in the edge (*edge-centrism* [GL+15]), locality plays a fundamental role, and global connectivity and centralisation fall short. However, to realize the FMEC vision, multiple challenges need to be addressed, including programmability, mobility support, resource management, service management, adaptivity, reliability, as well as security and privacy [RLM18; Shi+16].

In other words, the recurring theme in FMEC is the *smart exploitation* of resources, i.e., the *utilisation of idle resources* from devices that were not usually *fully* considered for computational or storage purposes (e.g., networking and end devices), and the *coordination of resources and tasks* to both extend the possibilities of individual components and attain non-functional advantage in system as well as user processes.

Computation offloading and cyber-foraging *Computation offloading* is the practice of delegating computation from one device to another. *CloneCloud* [Chu+11] was a prototype system that could automatically partition applications and migrate executions from a mobile device to the corresponding "clone" in the cloud. In *cyber foraging* [SKK12], devices look for offloading opportunities to *surrogates*, i.e., other nearby devices with idle resources. The process consists of five steps: (i) surrogate discovery; (ii) context monitoring, for local and

surrogate resource usage; (iii) task partitioning into sub-tasks, manually or automatically; (iv) offloading, i.e., scheduling of tasks on surrogate devices; and (v) remote execution control, exchanging control data with surrogates as well as inputs and outputs of the tasks. MAUI [Cue+10] enables fine-grained, energy-aware offloading of mobile code to remote servers, without requiring the programmer to deal with application partitioning and avoiding process/VM migration overhead by automatically deciding what methods should be executed remotely. *Scavenger* [Kri10] was another, Python-based, cyber-foraging system.

Mobile cloud computing (MCC) and cloudlets MCC aims to support development of mobile applications without the limitations of the computational resources of mobile devices. It does so by combining cloud computing, mobile computing, and wireless networks. In this paradigm, mobile devices become *thin clients* that *offload* computations to the cloud. Clouds can be distant (*remote clouds*), requiring WAN-connectivity, or nearby *cloudlets* [Sat+09; Ver+12], accessible through LAN-connectivity.

Volunteer computing In volunteer computing [And10], computer owners or users can provide their spare resources to various "projects". BOINC (Berkeley Open Infrastructure for Network Computing) [And04] is a well-known, open-source, client-server middleware for volunteer computing. Volunteer and cloud computing can be combined into *volunteer cloud computing* [CSD11], whereby volunteered resources are consolidated into a cloud. *cuCloud* [Men+18] is a Volunteer Computing-as-a-Service system implemented with Apache CloudStack (an open-source cloud management platform). *Nebula* [CWH13] is a cloud service that uses volunteer edge resources for geographically distributed data-intensive computing (i.e., for applications that require large amounts of data). It has a web front-end node for users to join the system as volunteers or executing applications. Each Nebula application has a master node for handling computations and a master node for dealing with data storage, each controlling a group of volunteer nodes. Related to volunteer clouds is the notion of *community clouds* [Kha+13], proposed as an extension of community networks [Jim+13].

Mobile (edge-)clouds A *mobile cloud* or *edge cloud* [Dro+13; MCC12] is a collection of mobile edge devices that aggregate their resources and make them available through a network. In this vision, devices are *thick clients* that offer idle resources for task offloading. Efforts in this direction include Hyrax [Teo12], Mobile Device Cloud [MHF13], Mobile Compute Cloud (MC^2) [Jai+13], Serendipity [Shi+12], femto-clouds [Hab+15]. Of course, this notion of edge-cloud can leverage volunteer computing and cyber-foraging.

Peer-to-peer edge-clouds A specialised version of edge-clouds is given by fully decentralised, *peer-to-peer clouds* [BMT12]. In this context, focus is largely on algorithms enabling direct offloading of computations to nearby devices. For instance, Honeybee [FLR16] is a work-sharing model for independent jobs that addresses dynamism through opportunistic computing. Mycocloud [Dub+15] is another algorithm – bio-inspired, self-organising – for service placement in decentralised clouds.

6.4 Application Development and Deployment on Complex Infrastructure

6.4.1 Microservices

A *microservice* is a software service that is:

- *small and focussed* — i.e., it follows the *single responsibility principle*; and
- *independent* — i.e., it can be independently developed and deployed.

The point, however, is not an individual service, but rather the implications of engineering *systems of microservices*. According to [Bon16], a *microservices-based architecture*

> *advocates creating a system from a collection of small, isolated services, each of which owns their data, and is independently isolated, scalable, and resilient to failure.*

That is, microservice-oriented development fosters isolation (adding, as a side effect, some burden to devops), in order to support agility in development and

operations by overcoming the issues of monolithic architectures (where an application comes as a single, large component that must be deployed and operated as a whole).

6.4.2 Cloud-native computing

Cloud-native computing [Cnc] is a paradigm that leverages techniques and technologies like microservices, containers, and automation for building applications that can be seamlessly run and operated in dynamic, cloud-like environments. *Cloud-native applications*, so, are essentially *"loosely coupled systems that are resilient, manageable, and observable"* [Cnc] and, as a consequence, can be variously operated. A well-known set of principles for building cloud-native applications is the *12-Factor App methodology* [12f], by Heroku.

6.4.3 Elasticity

The concept of *elasticity* in systems [MCD18] aims to convey the principle of flexible system operation: by application or removal of forces, there is a corresponding stretching or shrinking of balancing forces. Accordingly, ecosystems of people, processes, and things are dynamically managed considering both infrastructural and application requirements—constantly trading-off multiple dimensions (e.g., resources, quality, cost).

Osmotic Computing [Mas+16] — In Osmotic Computing, the osmosis metaphor is used to denote the opportunistic deployment of containerised microservices to the edge and/or to the cloud in order to balance contrasting needs while respecting constraints. A main challenge is optimising deployment with respect to multiple criteria to support effective decision-making for both short-term and long-term needs. This approach is used, e.g., in [RDR18] to regulate diffusion and relocation of brokers and clients across edge-cloud environments through a "pulling effect" based on a notion of "osmotic pressure" that aggregates a quantification of proximity and demand and that needs to be balanced by the osmotic controller.

6.5 Application Development and Deployment for Ad-Hoc Systems

The opposite scenario to complex, rich, infrastructure is when little or no infrastructure is available. Networks that do not rely on pre-existing infrastructure (e.g., routers, base stations) are called *ad hoc*. The research field of *Mobile Ad-hoc Networks (MANETs)* [CG14] studies how nodes interacting in a decentralised, ad hoc (or peer-to-peer) fashion while moving in the environment can self-organise to support networking functionality and applications. This is related to *spontaneous networking* [FAW01], *peer-to-peer systems* [SW05], and *wireless sensor networks* [MP11].

mobile ad-hoc networks

peer-to-peer systems

In this thesis, the focus is primarily on the software engineering perspective rather than on the networking perspective. Approaches that support programming of MANETs are surveyed in Chapters 3 and 4. The approach covered in this thesis, introduced in Chapter 5, can work in both ad-hoc and infrastructure-based settings, as shown in Chapter 9.

6.6 Final Remarks

Modern ICT infrastructures are complex, but research is working on tools that attempt to resolve such a complexity through abstraction layers. A key point of this work is that elasticity, intelligent deployment, and resource exploitation in edge/cloud environments have to be enabled by declarative programming models such as those covered in Chapter 4 and Chapter 5. In particular, the problem of finding efficient execution strategies can be delegated to platforms and middlewares, with applications providing just the domain logic and possibly *hints* to guide application partitioning or help controllers in decision-making. Another, related challenge revolves around sustaining operation when infrastructure falls down or supporting it in the first place when infrastructure is absent. Chapter 9 describes a proof-of-concept middleware for aggregate computations that supports multiple execution strategies for diverse infrastructural setups. Chapter 10 describes a pattern that could be useful for decentralised coordination of devices in

edge-clouds.

References

[12f] *12 Factor App Methodology website*. `https://12factor.net`. Retrieved October 28-th 2019. 2019.

[And04] David P Anderson. "Boinc: A system for public-resource computing and storage". In: *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. IEEE. 2004, pp. 4–10.

[And10] David P Anderson. "Volunteer computing: the ultimate cloud." In: *ACM Crossroads* 16.3 (2010), pp. 7–10.

[BMT12] Ozalp Babaoglu, Moreno Marzolla, and Michele Tamburini. "Design and implementation of a P2P Cloud system". In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM. 2012, pp. 412–417.

[Bon+12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. "Fog computing and its role in the internet of things". In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM. 2012, pp. 13–16.

[Bon16] Jonas Bonér. "Reactive microservices architecture: design principles for distributed systems". In: (2016).

[CG14] Marco Conti and Silvia Giordano. "Mobile ad hoc networking: milestones, challenges, and new research directions". In: *IEEE Communications Magazine* 52.1 (2014), pp. 85–96. DOI: `10.1109/MCOM.2014.6710069`. URL: `https://doi.org/10.1109/MCOM.2014.6710069`.

[Chu+11] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. "Clonecloud: elastic execution between mobile device and cloud". In: *Proceedings of the sixth conference on Computer systems*. ACM. 2011, pp. 301–314.

[Cnc] *Cloud Native Computing Foundation website*. `https://www.cncf.io`. Retrieved October 28-th 2019. 2019.

[CSD11] Fernando Costa, Luis Silva, and Michael Dahlin. "Volunteer cloud computing: Mapreduce over the internet". In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE. 2011, pp. 1855–1862.

[Cue+10] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. "MAUI: making smartphones last longer with code offload". In: *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM. 2010, pp. 49–62.

[CWH13] Abhishek Chandra, Jon Weissman, and Benjamin Heintz. "Decentralized edge clouds". In: *IEEE Internet Computing* 5 (2013), pp. 70–73.

[DB16] Amir Vahid Dastjerdi and Rajkumar Buyya. "Fog computing: Helping the Internet of Things realize its potential". In: *IEEE Computer* 49.8 (2016), pp. 112–116.

[Dro+13] Utsav Drolia, Rolando Martins, Jiaqi Tan, Ankit Chheda, Monil Sanghavi, Rajeev Gandhi, and Priya Narasimhan. "The case for mobile edge-clouds". In: *Ubiquitous Intelligence and Computing, 2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC)*. IEEE. 2013, pp. 209–215.

[Dub+15] Daniel Dubois, Giuseppe Valetto, Donato Lucia, and Elisabetta Di Nitto. "Mycocloud: Elasticity through self-organized service placement in decentralized clouds". In: *Int. Conf. on Cloud Computing (CLOUD)*. IEEE. 2015, pp. 629–636.

[EPM13] Thomas Erl, Ricardo Puttini, and Zaigham Mahmood. *Cloud computing: concepts, technology & architecture*. Pearson Education, 2013.

[FAW01] Laura Marie Feeney, Bengt Ahlgren, and Assar Westerlund. "Spontaneous networking: an application oriented approach to ad hoc networking". In: *IEEE Communications magazine* 39.6 (2001), pp. 176–181.

[FLR13] Niroshinie Fernando, Seng W Loke, and Wenny Rahayu. "Mobile cloud computing: A survey". In: *Future Generation Computer Systems* 29.1 (2013), pp. 84–106.

[FLR16] Niroshinie Fernando, Seng Loke, and Wenny Rahayu. "Computing with nearby mobile devices: a work sharing algorithm for mobile edge-clouds". In: *IEEE Transactions on Cloud Computing* 1 (2016), pp. 1–1.

[GL+15] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, et al. "Edge-centric computing: Vision and challenges". In: *ACM SIGCOMM Computer Communication Review* 45.5 (2015), pp. 37–42.

[GYL11] Raghu K Ganti, Fan Ye, and Hui Lei. "Mobile crowdsensing: current state and future challenges". In: *IEEE Communications Magazine* 49.11 (2011).

[Hab+15] Karim Habak, Mostafa Ammar, Khaled A Harras, and Ellen Zegura. "Femto clouds: Leveraging mobile devices to provide cloud service at the edge". In: *2015 IEEE 8th international conference on cloud computing*. IEEE. 2015, pp. 9–16.

[Hu+15] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. "Mobile edge computing—A key technology towards 5G". In: *ETSI white paper* 11.11 (2015), pp. 1–16.

[Ign16] Roger Ignazio. *Mesos in action*. Manning Publications Co., 2016.

[Jai+13] P Jain, R Kabra, S Rustagi, T Bansal, D Patel, and V Raychoudhury. "MC 2: on-the-fly mobile compute cloud for computational intensive task". In: *Proceedings of the 5th IBM Collaborative Academia Research Exchange Workshop*. ACM. 2013, p. 7.

[Jim+13] Javi Jiménez, Roger Baig, Pau Escrich, Amin M Khan, Felix Freitag, et al. "Supporting cloud deployment in the Guifi.net community network". In: *Global Information Infrastructure Symposium, 2013*. IEEE. 2013, pp. 1–3.

[Kha+13] Amin M Khan, Leandro Navarro, Leila Sharifi, and Luís Veiga. "Clouds of small things: Provisioning infrastructure-as-a-service from within community networks". In: *Wireless and Mobile Computing, Networking and Communications (WiMob), 9th Int. Conf. on*. IEEE. 2013, pp. 16–21.

[Kri10] Mads Darø Kristensen. "Scavenger: Transparent development of efficient cyber foraging applications". In: *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*. IEEE. 2010, pp. 217–226.

[Mas+16] Villari Massimo, Fazio Maria, Dustdar Schahram, Rana Omer, and Rajiv Ranjan. "Osmotic Computing: A New Paradigm for Edge/Cloud Integration". In: *IEEE Cloud Computing* 3.6 (2016), pp. 76–83. ISSN: 2325-6095. DOI: doi.ieeecomputersociety.org/10.1109/MCC.2016.124.

[MB17] Pavel Mach and Zdenek Becvar. "Mobile edge computing: A survey on architecture and computation offloading". In: *IEEE Communications Surveys & Tutorials* 19.3 (2017), pp. 1628–1656.

[MCC12] Emiliano Miluzzo, Ramón Cáceres, and Yih-Farn Chen. "Vision: mClouds-computing on clouds of mobile devices". In: *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM. 2012, pp. 9–14.

[MCD18] Daniel Moldovan, Georgiana Copil, and Schahram Dustdar. "Elastic systems: Towards cyber-physical ecosystems of people, processes, and things". In: *Computer Standards & Interfaces* 57 (2018), pp. 76–82.

[Men+18] Tessema M Mengistu, Abdulrahman M Alahmadi, Yousef Alsenani, Abdullah Albuali, and Dunren Che. "cucloud: Volunteer computing as a service (vcaas) system". In: *International Conference on Cloud Computing*. Springer. 2018, pp. 251–264.

[MG+11] Peter Mell, Tim Grance, et al. "The NIST definition of cloud computing". In: (2011).

[MHF13] Abderrahmen Mtibaa, Khaled A Harras, and Afnan Fahim. "Towards computational offloading in mobile device clouds". In: *2013 IEEE 5th international conference on cloud computing technology and science*. Vol. 1. IEEE. 2013, pp. 331–338.

[MP11] Luca Mottola and Gian Pietro Picco. "Programming wireless sensor networks: Fundamental concepts and state of the art". In: *ACM Computing Surveys (CSUR)* 43.3 (2011), p. 19.

[PG74] Gerald J Popek and Robert P Goldberg. "Formal requirements for virtualizable third generation architectures". In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

[RDR18] Thomas Rausch, Schahram Dustdar, and Rajiv Ranjan. "Osmotic message-oriented middleware for the internet of things". In: *IEEE Cloud Computing* 5.2 (2018), pp. 17–25.

[RLM18] Rodrigo Roman, Javier Lopez, and Masahiro Mambo. "Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges". In: *Future Generation Computer Systems* 78 (2018), pp. 680–698.

[SAE12] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. "OpenStack: toward an open-source solution for cloud computing". In: *International Journal of Computer Applications* 55.3 (2012), pp. 38–42.

[Sat+09] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. "The case for vm-based cloudlets in mobile computing". In: *IEEE Pervasive Computing* 8.4 (2009).

[Shi+12] Cong Shi, Vasileios Lakafosis, Mostafa H Ammar, and Ellen W Zegura. "Serendipity: enabling remote computing among intermittently connected mobile devices". In: *Proceedings of the thirteenth ACM international symposium on Mobile Ad Hoc Networking and Computing*. ACM. 2012, pp. 145–154.

[Shi+16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. "Edge computing: Vision and challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646.

[SKK12] Mohsen Sharifi, Somayeh Kafaie, and Omid Kashefi. "A survey and taxonomy of cyber foraging of mobile devices". In: *IEEE Communications Surveys & Tutorials* 14.4 (2012), pp. 1232–1243.

[SW05] Ralf Steinmetz and Klaus Wehrle. *Peer-to-peer systems and applications*. Vol. 3485. Springer, 2005.

[Teo12] Chye Liang Vincent Teo. "Hyrax: Crowdsourcing mobile devices to develop proximity-based mobile clouds". In: *Pittsburgh: Carnegie Mellon University* (2012).

[TR88] Andrew Stuart Tanenbaum and R van Renesse. “A critique of the remote procedure call paradigm”. In: (1988).

[Ver+12] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. “Cloudlets: Bringing the cloud to the mobile user”. In: *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM. 2012, pp. 29–36.

[VRM14] Luis M Vaquero and Luis Rodero-Merino. “Finding your way in the fog: Towards a comprehensive definition of fog computing”. In: *ACM SIGCOMM Computer Communication Review* 44.5 (2014), pp. 27–32.

[Yaq+16] Ibrar Yaqoob, Ejaz Ahmed, Abdullah Gani, Salimah Mokhtar, Muhammad Imran, and Sghaier Guizani. “Mobile ad hoc cloud: A survey”. In: *Wireless Communications and Mobile Computing* 16.16 (2016), pp. 2572–2589.

Part II

Contribution

Chapter 7

ScaFi: Aggregate Programming in Scala

The diversity of languages is not a diversity of signs and sounds but a diversity of views of the world.

Wilhelm von Humboldt

Contents

As covered in Chapter 5, aggregate computing is a paradigm for programming collective adaptive systems, embodied by languages like Proto [BB06] and Protelis [PVB15]. Both Proto and Protelis are standalone domain-specific languages (DSL) and, as a consequence, require an ad-hoc toolchain.

For a better integration of aggregate computing techniques into mainstream software development, as well as to investigate on embedding the field calculus in modern, object-oriented and functional languages, in this chapter we present ScaFi, an aggregate programming DSL internal to the Scala programming language. In order to seamlessly embed field computations into Scala, the design choice of *not* explicitly exposing field types has been made: this yielded a new field calculus variant, called FScaFi, which preserves basic expressivity and where explicit fields can still be provided through reification at the library level (cf. Section 7.5.2).

ScaFi is motivated and presented in Sections 7.1 and 7.2. FScaFi is described formally in Section 7.3 and Section 7.4. Finally, Sections 7.5 and 7.6 show ScaFi in action.

7.1 Motivation and Problem

7.1.1 Why ScaFi

A key issue for the applicability of the aggregate programming model is to find ways to smoothly integrate it with the standard development practice—languages, processes and tools.

Previously, a language for expressing field-based computations called Protelis [PVB15] was introduced: it was based on an untyped, *standalone DSL* (also called, *external DSL*), providing support to import existing Java libraries and call

methods on objects. A PROTELIS specification, then, feeds an interpreter that can be run on a device to execute local field computation rounds. This approach, though enabling Aggregate Computing in deployed contexts, might difficulty combine with the software development process one typically exploits to build complex, distributed applications. In particular, the following issues make the process unsmooth: training and documenting for a *distinct* language can be burdensome; extra development and maintenance effort is needed to adequately support editing tools (e.g., plugins are required for common *Integrated Development Environment (IDE)* features like syntax highlighting, validation, refactoring); activities that span both the DSL and the target language, such as static analysis and debugging, may be hard to implement; and finally, the ability to smoothly reuse the features and libraries of the target language can be limited. Though language development toolchains greatly improved recently (cf. the Xtext language workbench [Bet16] and its Xbase extension [Eff+12], to name a popular one), practically, with an external DSL it may be difficult to come up with a cohesive process and design for a target software system (cf. Generation gap pattern [Vli98]), since parts written in the DSL need to bidirectionally refer and interact with other parts of the system [Gho11].

A prominent, modern approach to address this problem is to devise an *internal DSL* [Voe13] (also called *embedded DSL* [Gho11]) that provides mechanisms to support the new features on top of an adequate and well-known host programming language. Of course, the syntax of the embedded DSL is limited by the constraints exerted by the host language; therefore, it is fundamental to take into account the requirements as well as the desiderata for the DSL, which in our case include:

- *pragmatism*—supporting easy reuse of existing programming structures and mechanisms;
- *reliability*—intercepting errors concerning syntax and types at compile-time;
- *expressivity*—offering a concise and eloquent syntax, minimising the accidental language complexity induced by the environment.

All these considerations led us to choose the Scala programming language as the host for what has become the SCAFI aggregate computing DSL and platform [CPV16; VCP16]. The goal with SCAFI is to provide an environment to

streamline and support effective development of systems based on the Aggregate Computing paradigm, leveraging the solid basis provided by a mainstream programming language such as Scala and its ecosystem. Moreover, it is found significant to integrate a novel research paradigm in the context of a widespread programming environment already supporting the core paradigms found in modern software development, namely Object-Oriented Programming (OOP) and Functional Programming (FP). In fact, Scala:

- runs on the JVM and thus enables straightforward interaction and reuse of libraries in the Java ecosystem;
- offers a strong, expressive type system, with type inference, that helps to prevent errors and also comes handy for building libraries (even allowing for type-level computations);
- has quite a flexible syntax that makes it possible for library designers to create elegant Application Program Interfaces (APIs).

The multi-paradigm and popularity aspects were also in favour of Scala (cf., imperative, object-oriented, functional, component-based programming support [OZ05; OR14]) with respect to languages also suitable for DSL development like Haskell or Racket.

Incidentally, Scala also has great popularity in the distributed computing arena: it is the implementation language for several distributed computing libraries and frameworks [Cal+17; Die+16], there including popular ones for streaming (Apache Kafka [Apaa]) and cluster computing (Apache Spark [Apab]). Additionally, it well supports linguistic abstractions for concurrent and distributed computing by the Akka actor framework [Akk], which is an industry-ready solution for implementing resilient, message-driven runtimes and applications. Hence, our choice of Scala also fosters the construction of a platform-level support on top of ScaFi (see Chapter 9), in the form of a middleware for running distributed and situated systems.

As will be showcased in the next section, the flexibility of Scala allows us to provide smooth field calculus support by: *(i)* a cohesive syntax to handle field expressions (which are actually standard expressions), *(ii)* mechanisms to control *when* and *how* expressions are evaluated (thanks to by-name parameters), and *(iii)*

declaratively handling neighbour interactions with a new notion of *computation "against" a neighbour*, namely, a computation whose outcome depends on the most recent, local view of the result of computation in that neighbour (differently from standard field calculus, this allows smooth application of host typing mechanisms to any field expression).

7.1.2 Embedding field computations in a host language

A first key problem in embedding field computations in a host language lies in the potential mismatch between the local representation of types and the representation of field expressions. For the former, one would seek for the natural representation of the host language, e.g., literal `1` representing a local value of type `Int`. For the latter, one has to combine the natural representation (since field expressions include local expressions as described above) with the additional constructs manipulating fields; namely, literal `1`, or some variation, should represent a field value (equal to 1 in all space-time points of the local device's neighbourhood) of some type, say `Field[Int]`, inheriting all the operations one would use over the local counterpart `Int` (`+`, `-`, and so on). As a consequence, standard types such as numbers, booleans, characters, objects, and the corresponding operators, need to be coherently lifted in order to work under the field interpretation. That is, given any type `T`, the type `Field[T]` should support all the operations provided by `T` but lifted to the field context (cf., functors in category theory): for instance, expression $\texttt{e}_1 + \texttt{e}_2$ where $\texttt{e}_1$ and $\texttt{e}_2$ have type `Field[Int]` should naturally give an element of type `Field[Int]`. Ideally, the type system should continue to do its job with field types, and field expressions should be written in a notation which is analogously simple as the local counterpart.

A second key problem stems from the declarativeness and compositionality of field computations: the host language should provide the means for defining blocks of code and for controlling *when* and *how* these are executed, mainly by deferring their evaluation until a later time and properly contextualising them to the local information available in each device. These mechanisms should also be lightweight so as to keep the impact on the user-side of the DSL as little as possible.

The third, key technical difficulty is to properly deal with the interaction model of field computations, which is neighbour-driven. Field computations are equipped

with a (logical or physical) notion of *neighbourhood* that basically expresses the boundary of direct communication, i.e., what devices can be reached by a given device through a communication act occurring in a certain position of computation. The observation of values that a device reads from its neighbours is typically modelled by reification into a *neighbouring value* (a map from neighbours to values), which can be manipulated functionally until being collapsed back to a local value by means of some *folding* operator—e.g., computing the minimum value. This requires one to explicitly differentiate, syntactically and semantically, the two classes of types, neighbouring types and local types, raising the issue of how to lift standard local operators to neighbour types. Thanks to the features of the Scala programming language, and as detailed in this chapter, ScaFi handles this problem with a twist:

- the same type, say `Int`, is used both for local types and neighbouring types;
- the notion of computation over neighbouring values is semantically turned into a notion of computation "against" a neighbour (namely, a computation whose outcome depends on recent result of computation in that neighbour), hence there is no longer need of two kinds of type;
- folding operations are the triggers for a universal quantification process, iterating computations against all pertinent neighbours.

Such changes, impacting theory and practice of field computations as described in this chapter, enable expressive and smooth integration with Scala programming.

7.2 Computational Fields in Scala

In Scala, every value is an object and behaviour ultimately resides in methods: thus, in order to embed field computations in Scala, field operators have to become methods in some object which is responsible for their interpretation. That is, programming by the DSL means calling the methods exposed by the field constructs API, which are implemented by an interpreter object; such an object, assumed to be accessible, is a sort of local virtual machine for field computations, which also provides access to the local execution context (i.e., where critical information about the specific device computation are available). A field

computation is then carried out locally as a combination of method calls that transparently and recursively build an internal data structure (a tree of values), representing the local result of computation, and out of which the content of the message to be sent to neighbours is derived. This computational mechanism, along with others to manage field evolutions and communication across neighbours via neighbour-dependent expressions, is what is implemented by ScaFi as described in this section and modelled by the Featherweight ScaFi (FScaFi) calculus introduced in Section 7.3.

7.2.1 Constructs

The following interface, implemented as a Scala trait, represents the basic field computation constructs as methods:

```
trait Constructs {
  // Key constructs
  def rep[A](init: => A)(fun: (A) => A): A
  def foldhood[A](init: => A)(aggr: (A, A) => A)(expr: => A): A
  def nbr[A](expr: => A): A
  def @@[A](b: => A): A

  // Abstract types
  type ID         // type of device identifiers
  type LSNS, NSNS // type of local and neighbour sensors

  // Contextual, but foundational
  def mid(): ID
  def sense[A](name: LSNS): A
  def nbrvar[A](name: NSNS): A
}
```

In Scala, methods are introduced with the `def` keyword, can be generic (with type parameters specified in square brackets), may accept multiple parameter lists, and specify a return type at the end of the signature (when this is not given the compiler attempts inference). Function types may take the form `(`I_1`,...,`I_N`)=>`O, which is actually syntactic sugar over `FunctionN[`I_1`,...,`I_N`,`O`]`; curried function types can be written as I_1`=>`$\cdots$`=>`I_N`=>`O (`=>` is right associative). Tuple types may take the form `(`T_1`,...,`T_N`)`, which is actually syntactic sugar over `TupleN[`T_1`,...,`T_N`]`; similarly, a literal tuple value can be denoted as `(`v_1`,...,`v_N`)`. By-name parameters, denoted with type `=>`T, capture expressions or blocks of

code that are passed unevaluated to the method and are actually evaluated every time the parameter is used—they are basically syntactic sugar over 0-ary function types. As a relevant note on syntax, especially useful in DSLs to render constructs with code blocks, unary parameter lists in a method can be called also with curly brackets instead of parentheses. E.g., all the following are valid invocations for `rep` method above: `rep(·)(·)`, `rep(·){·}`, `rep{·}(·)`, `rep{·}{·}`. Finally, nullary methods can be invoked without parentheses; e.g., `mid` is a valid method call just like `mid()`.

First of all, note that method signatures do not include field-like type constructors. In fact in ScaFi, fields are not reified explicitly but only exist at the semantic level, namely, a Scala expression is handled as a field expression when passed to the ScaFi interpreter. Accordingly, one can adopt two useful viewpoints at aggregate specifications: the *local* viewpoint, typically useful when reasoning about low-level aspects of field computations, which considers a field expression as the computation carried on by a specific individual device; and the *global* viewpoint, typically more useful when focussing on higher-level composition of field computations, which regards a specification at the aggregate level, as a whole spatio-temporal computation evolving a field. So, an expression of a given Scala type (say `Int` or `List[Double]`) can represent the outcome of execution of a computation locally (an `Int` or `List[Double]`), or globally as the program producing a field (a field of `Int`s or a field of `List[Double]` values). As an example, field expression `1` can be locally seen as a device producing a 1 at a certain round, or globally as a "flat" field holding a 1 at each space-time event (i.e., in any round of any device).

Considering the local interpretation, two key concepts need to be clarified before diving into further details. First, when evaluating a given (sub-)expression in a device d, a neighbour device is said to be *aligned* if, at its latest round, it evaluated the same subexpression: in fact, because of branching mechanisms, devices can evaluate different parts of the main expression representing the overall field, hence in general the set of aligned neighbours is smaller than the actual neighbourhood, which always includes the device d itself. Second, a (sub-)expression is said to be *neighbour-dependent* if its evaluation will be performed "against" an aligned neighbour n, such that the outcome will depend on the outcome of the

evaluation of the same expression on n as occurred at its latest round: we will see that this can happen because of constructs `nbr` and `nbrvar`, whose result is in fact a neighbour-dependent one. Note that globally, a neighbour-dependent expression can be seen as generating a field associating each space-time point with a *neighbouring value*, i.e., a value obtained by evaluating the expression against an aligned neighbour device.

We now describe each construct in turn, elucidating both its local and global interpretation, before full examples will be provided in Section 7.2.2:

- Construct `rep(init)(f)` provides the (only) means for evolving fields over time: globally, the output field is obtained by continuously applying the state transformation function field `f` throughout space and time on a field which, at the beginning, is `init`; locally, instead, one can interpret the expression as the result yielded by a device upon application of the unary transition function `f` to the value computed at the previous round, or to the local value of `init` at the first round. As we will see, `rep(0)(_+1)`[1] is the field counting the number of rounds executed at each device.
- Construct `foldhood(init)(acc)(e)` (where `acc` is a binary accumulator function and `init` is a terminal value—the two typically forming a monoidal structure[2]), provides a way of extracting information from neighbours: globally, it yields a field of local values obtained by everywhere and everytime collapsing (i.e., folding through `acc` and `init`) the field of neighbour values defined by `e`, which may possibly be neighbour-dependent; locally, it evaluates the expression `e` against each different aligned neighbour, and the set of results feeds a purely functional folding process using `acc` and `init`. As we will see, `foldhood(0)(_+_){1}` is the field dynamically counting the number of neighbours of each device (since 1 is not neighbour-dependent, it simply sums a 1 per neighbour).
- Construct `nbr(e)` defines a neighbour-dependent expression used to support interaction across neighbours: globally, it creates a field of neighbouring val-

[1] In Scala, underscores `_` provide syntactic sugar for creating lambdas, by representing how subsequent parameters are used in expressions being the body of such lambdas, such that, e.g., `f(_,_)` is like `_1,_2=>f(_1,_2)` and `_+x` like `_1=>_1+x`.

[2] I.e., typically, `acc` is associative, and `init` is an identity for `acc`, though these properties are not strictly required.

ues associating to each device d at each round a map from aligned neighbours to the most recent values of `e` they evaluated; locally, when executed against a neighbour device n, it gives the most recent value of `e` computed by n. As we will see, `foldhood(0)(_+_){nbr{`e`}}` is the field mapping across time each device to the sum of evaluations of e across neighbours.

- Construct `@@(b)` wraps the body `b` of a standard Scala function so as to make it an *aggregate function*, namely, a "unit of alignment": globally, when an aggregate function is called, the space-time is split in regions executing the same body and, in each such region, computation gets isolated from others by branching; locally, this is achieved by excluding from the set of aligned neighbours those which are executing a different aggregate function body. Note this construct is needed in ScaFi API to properly support alignment.

Other operators that do not affect space-time behaviour but are somehow foundational include:

- construct `sense(lsns)`, to query a local sensor of name `lsns`: this is the mechanisms by which an expression can interact with the local context to receive information from the physical world (temperature, humidity, and so on) or the platform (GPS position, time elapsed since latest round, and so on);
- construct `mid()`, which is a particular sensor returning the unique identifier of the running device;
- construct `nbrvar(nsns)`, which can be used to query a "neighbouring sensor" of name `nsns` yielding a (field of) neighbouring value(s): a typical such sensor is `nbrRange`, used to ask the platform to estimate physical distances to neighbours.

7.2.2 Examples

Here, we incrementally describe some example applications of the above constructs, to clarify some details of the language before its formalisation in Section 7.3 as well as to pave the way towards a more complex case study as described in Section 7.6. Unless differently specified, the following descriptions shall rely on the global stance.

In ScaFi, a usual literal such as, for instance, tuple

```
("hello", 7.7, true)
```

is to be seen as a *constant* (i.e., not changing over time) and *uniform* (i.e., not changing across space) field holding the corresponding local value at any point of the space-time domain. By analogy, an expression such as

```
1 + 2
```

denotes a global expression where a field of 1s and a field of 2s are summed together through the field operator `+`, which works like a point-wise application of its local counterpart. Indeed, literal `+` can also be thought of as representing a constant, uniform field of (binary) functions, and function application can be viewed as a global operation that applies a function field to its argument fields.

A constant field does not need to be uniform. For instance, given a static network of devices, then

```
mid()
```

denotes the field of device identifiers, which does not change across time but does vary in space. On the other hand, expression

```
sense("temperature") // or sense[Double]("temperature") to explicitly type it
```

is used to represent a field of temperatures (as obtained by collectively querying the local temperature sensors over space and time), which is in general non-constant and non-uniform.

Fields changing over time can also be programmatically defined by the `rep` operator; for instance, expression

```
// Initially 0; state is incremented at each round
rep(0){ x => x + 1 } // Equally expressed in Scala as: rep(0)(_ + 1)
```

counts how many rounds each device has executed: it is still a non-uniform field since the update phase and frequency of the devices may vary both between devices and across time for a given device.

Folding can be used to trigger the important concept of neighbour-dependent expression. As a simple initial example, expression

```
foldhood(0)(_ + _){ 1 }
```

counts the number of neighbours at each device (possibly changing over time if the network topology is dynamic). Note that folding collects the result of the evaluation of 1 against all neighbours, which simply yields 1, so the effect is merely the addition of 1 for each existing neighbour.

The key way to define truly neighbour-dependent expressions is by the `nbr` construct, which enables to "look around" just one step beyond a given locality. Expression

```
foldhood(0)(_ + _){ nbr { sense[Double]("temperature") } } / foldhood(0)(_ + _){
    1 }
```

evaluates to the field of average temperature that each device can perceive in its neighbourhood. The numerator sums temperatures sensed by neighbours (or, analogously, it sums the neighbour evaluation of the temperature sensor query expression), while the denominator counts neighbours as described above.

As another example, the following expression denotes a Boolean field of warnings:

```
val warningThreshold: Double = 42.0

foldhood(false)(_ || _){
  nbr { sense[Double]("temperature") } > warningThreshold
}
```

This is locally `true` if any neighbour perceives a temperature higher than some topical threshold. Notice that by moving the comparison into the `nbr` block,

```
foldhood(false)(_ || _){
  nbr { sense[Double]("temperature") > warningThreshold }
}
```

the decision about the threshold (i.e., the responsibility of determining when a temperature is dangerous) is transferred to the neighbours, and hence warnings get blindly extended by 1-hop. Of course, provided `warningThreshold` is uniform, the result would be the same in this case.

Ordinary Scala functions can be defined to capture and give a name to common field computation idioms, patterns, and domain-specific operations. For instance,

by assuming a `mux` function that implements a strictly-evaluated version of `if`:

```
def mux[A, B<:A, C<:A](cond: Boolean)(th: B)(el: C): A = if(cond) th else el
```

a variation of `foldhood`, called `foldhoodPlus`[3], which does not take "self" (the current device) into account, can be implemented as follows:

```
def foldhoodPlus[A](init: => A)(aggr: (A, A) => A)(expr: => A): A =
  foldhood(init)(aggr)(mux(mid==nbr{mid}){ init }{ expr })
```

Notice that the identity `init` is used when considering a neighbour device whose identifier (`nbr{mid}`) is the same as that of the current device (`mid`). As another example, one can give a label to particular sensor queries, such as:

```
def temperature = sense[Double]("temperature")
def nbrRange = nbrvar[Double]("nbr-range")
```

The second case uses construct `nbrvar`, which is a neighbouring sensor query operator providing, for each device, a sensor value for each corresponding neighbour: e.g., for `nbrRange`, the output value is a floating-point number expressing the estimation of the distance from the currently executing device to that neighbour—so, it is usually adopted as a metric for "spatial algorithms". Based on the above basic expressions, one can define a rather versatile and reusable building block of Aggregate Programming, called *gradient* [LK87; Bea+08; Aud+17]. A gradient (see Figure 7.1) is a numerical field expressing the minimum distance (according to a certain `metric`) from any device to `source` devices; it is also interpretable as a surface whose "slope" is directed towards a given source. In ScaFi, it can be programmed as follows:

```
def gradient(source: Boolean, metric: () => Double = nbrRange): Double =
  rep(Double.PositiveInfinity){ distance =>
    mux(source) {
      0.0
    }{
      foldhoodPlus(Double.PositiveInfinity)(Math.min(_,_)){ nbr{distance} +
    metric }
    }
  }
```

[3]The "Plus" suffix is to mimic the mathematical syntax R^+ of the transitive closure of a (neighbouring) relation R.

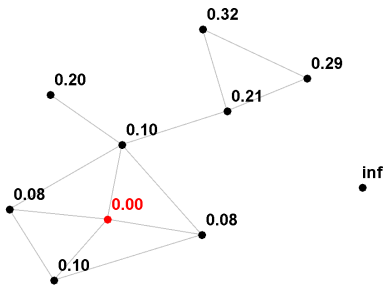


Figure 7.1: Pictorial representation of a gradient field snapshot in the midst of a simulation in ScaFi. The red nodes are the sources of the gradient. The nodes at the top-left have parted from the network and their values increase unboundly. The grey lines represent device connectivity according to a proximity-based neighbouring relationship.

The `rep` construct allows one to keep track of the distances across rounds of computations: source devices are at a null distance from themselves, and the other devices take the minimum value among those of neighbours increased by the corresponding estimated distances as given by `metric`—defaulting to `nbrRange`. Notice that a version of `foldhood` that does not consider the device itself must be used to prevent devices from getting stuck to low values because of self-messages (as it would happen when a source node that gets deactivated): with it, gradients dynamically adapt to changes in network topology or position/number of sources, i.e., it is self-stabilising [Vir+18].

Another common and important operation on fields is splitting computation into completely separate parts or sub-computations executed in isolated space-time regions. An example is computing a `gradient` in a space that includes obstacle nodes so that gradient slopes circumvent the obstacles: this is typically needed when the resulting structure is used for "navigation" towards the source, and one wants to avoid navigation to get stuck on a path interrupted by obstacles. The technical issue, here, is to prevent obstacle nodes to participate in gradient construction and share a distance that could be wrongly selected by some device. Note that the following erroneous code

```
mux(isObstacle){  Double.PositiveInfinity }{ gradient(isSource) } // erroneous
```

would first create the gradient in the entire space (because of strict evaluation),

and *then* set the obstacle nodes to `Double.PositiveInfinity`. The solution to the problem needs to leverage aggregate functions, and their ability of acting as units of alignment. That is, we can use a different 0-ary aggregate function for normal and obstacle nodes:

```
(mux(isObstacle){
  () => @@{ Double.PositiveInfinity }
}{
  () => @@{ gradient(isSource) }
})()
```

Calling such functions effectively restricts the domain to the set of devices executing them, thanks to the space-time branching enacted by construct `@@` wrapping *the bodies* of the corresponding Scala literal functions; by calling them exclusively in any device, the system gets partitioned into two sub-systems, each one executing a different sub-computation. For convenience, ScaFi provides as built-in function, called `branch`, defined as:

```
def branch[A](cond: => Boolean)(th: => A)(el: => A): A =
  mux(cond)(() => @@{ th })(() => @@{ el })()
```

With it, a gradient overcoming an obstacle is properly written as

```
branch(isObstacle){ Double.PositiveInfinity }{ gradient(isSource) } // correct
```

which is cleaner and hides some complexity while better communicating the intent: branching computation[4]. Generally, notation `@@` has to be used for bodies of literal functions that include aggregate behaviour, i.e., functions which (directly or indirectly) call methods of the `Constructs` trait—other uses have no effects on the result of computation.

We remark that the above field calculus expression (gradient avoiding obstacles) effectively creates a distributed data structure that is rigorously self-adaptive [Vir+18]: independently of the shape and dynamics of obstacle area(s), source area(s), metric and network structure, it will continuously bring about formation of the correct gradient, until eventually stabilising to it. Additionally, it can be used as building block for more complex applications, to which the self-adaption properties will be transferred, by simple functional composition.

[4]Notice that `if` construct of Scala cannot be used to branch over aggregate behaviour, hence either `mux` or `branch` should be used to control the alignment process.

7.3 FScaFi Calculus: Syntax and Semantics

We now move to the semantic details of the proposed model, which amounts to how an aggregate specification turns into computational rounds for devices and messages to be exchanged among devices. This section addresses the problem formally, presenting FEATHERWEIGHT SCAFI *(*FSCAFI*)*, a minimal core calculus that models the aggregate computing aspects of SCAFI—e.g., much as FJ [IPW01] models the object-oriented aspects of Java. The formalisation of FSCAFI is inspired by the formalisation of the *higher-order field calculus (HFC)* [Aud+19]—a thoughtful comparison between FSCAFI and HFC is presented in Section 7.4.

Devices undergo computation in rounds. When a round starts, the device gathers information about messages received from neighbours (only the last message from each neighbour is actually considered), performs an evaluation of the program, and finally emits a message to all neighbours with information about the outcome of computation. The scheduling policy of such rounds is abstracted in this formalisation, though it is typically considered fair and non-synchronous.

Section 7.3.1 presents the syntax of FSCAFI; Section 7.3.2 presents its type system; Section 7.3.3 presents an operational semantics for the computation that takes place on individual devices; and Section 7.3.4 presents an operational semantics for the evolution of whole networks.

7.3.1 Syntax

FSCAFI is a core subset of SCAFI, strictly retaining its syntax (i.e., modulo minor adjustments, FSCAFI expressions are valid SCAFI/Scala code). The syntax of FSCAFI is given in Figure 7.2. Following [IPW01], the overbar notation denotes metavariables over sequences and the empty sequence is denoted by $\bullet$; e.g., for expressions, we let $\overline{\mathsf{e}}$ range over sequences of expressions, written $\mathsf{e}_1, \mathsf{e}_2, \ldots \mathsf{e}_n$ ($n \geq 0$). FSCAFI focusses on aggregate programming constructs. In particular:

- it neglects the many orthogonal Scala features that one can use (object-oriented constructs, and the like), and
- it is parametric in the built-in data constructors and functions—in the examples, we consider the set of built-in data constructors and functions listed,

$\texttt{P} ::= \overline{\texttt{F}}\ \texttt{e}$	program
$\texttt{F} ::= \texttt{def d}(\overline{\texttt{x}})\ \texttt{= @@\{e\}}$	function declaration
$\texttt{e} ::= \texttt{x} \mid \texttt{v} \mid (\overline{\texttt{x}})\ \overset{\tau}{\texttt{=>}}\ \texttt{@@\{e\}} \mid \texttt{e}(\overline{\texttt{e}}) \mid \texttt{rep(e)\{e\}} \mid \texttt{nbr\{e\}} \mid \texttt{foldhood(e)(e)\{e\}}$	expression
$\texttt{v} ::= \texttt{c}(\overline{\texttt{v}}) \mid \texttt{f}$	value
$\texttt{f} ::= \texttt{b} \mid \texttt{d} \mid (\overline{\texttt{x}})\ \overset{\tau}{\texttt{=>}}\ \texttt{@@\{e\}}$	function value

Figure 7.2: Syntax of FScaFi.

with their types, in Section 7.3.2.

Note that—apart from specific Scala syntax—the examples of ScaFi code given in Section 7.2.2 are actually examples of FScaFi code. In particular, in order to turn functions `foldhoodPlus`, `gradient` and `branch` into FScaFi functions it is enough to drop type annotations and the default value for the parameter `metric` of `gradient`. To distinguish with respect to actual ScaFi code, in writing FScaFi code we do not provide syntax colouring.

A program `P` consists of a sequence $\overline{\texttt{F}}$ of function declarations and a main expression `e`. A function declaration `F` defines a (possibly recursive) function; it consists of a name `d`, $n \geq 0$ variable names $\overline{\texttt{x}}$ representing the formal parameters, and an expression `e` representing the body of the function.

Expressions `e` are the main entities of the calculus, modelling a whole field computation. An expression can be: a variable `x`, used as function formal parameter; a value `v`; an anonymous function $(\overline{\texttt{x}})\ \overset{\tau}{\texttt{=>}}\ \texttt{@@\{e\}}$ (where $\overline{\texttt{x}}$ are the formal parameters, `e` is the body, and τ is an internal *tag*); a function call $\texttt{e}(\overline{\texttt{e}})$; a `rep`-expression `rep(e){e}`, modelling time evolution; an `nbr`-expression `nbr{e}`, modelling neighbourhood interaction; or a `foldhood`-expression `foldhood(e)(e){e}` which combines values obtained from neighbours.

Tags τ of anonymous functions $(\overline{\texttt{x}})\overset{\tau}{\texttt{=>}}\ \texttt{@@\{e\}}$ do not occur in source programs: when the evaluation starts each anonymous function expression $(\overline{\texttt{x}})$ `=> @@{e}` occurring in the program is given a distinguished tag τ—two occurrences of the same anonymous function expression get different tags. In the following we will use the phrase *name of a function* to refer both to the tag of an anonymous function, or to the name of a built-in or declared function. As we will see below, names are

used to define the semantics of function call.

The set of the *free variables* of an expression `e`, denoted by $\mathbf{FV}(\mathtt{e})$, is defined as usual (the only binding construct is $(\overline{\mathtt{x}}) \overset{\tau}{\Rightarrow} \mathtt{@@\{e\}}$). An expression `e` is *closed* if $\mathbf{FV}(\mathtt{e}) = \bullet$. The main expression of any program must be closed.

A value can be either a *data value* $\mathtt{c}(\overline{\mathtt{v}})$ or a *functional value* `f`. A data value consists of a *data constructor* `c` of some arity $m \geq 0$ applied to a sequence of m data values $\overline{\mathtt{v}} = \mathtt{v}_1, ..., \mathtt{v}_m$. A data value $\mathtt{c}(\mathtt{v}_1, ..., \mathtt{v}_m)$ is written `c` when $m = 0$. According to the data constructors listed in Figure 7.4, examples of data values are: the Booleans `True` and `False`, numbers, pairs (like $\mathtt{Pair}(\mathtt{True}, \mathtt{Pair}(5, 7))$) and lists (like $\mathtt{Cons}(3, \mathtt{Cons}(4, \mathtt{Null}))$).

Functional values `f` comprise:

- declared function names `d`;
- closed anonymous function expressions $(\overline{\mathtt{x}}) \overset{\tau}{\Rightarrow} \mathtt{@@\{e\}}$ (i.e., such that $\mathbf{FV}(\mathtt{e}) \subseteq \{\overline{\mathtt{x}}\}$);
- built-in functions `b`, which can in turn be:
  - *pure operators* `o`, such as functions for building and decomposing pairs (`pair`, `fst`, `snd`) and lists (`cons`, `head`, `tail`), the `mux` function, the equality function (=), mathematical and logical functions (`+`, `&&`, ...), and so on;
  - *sensors* `s`, which depend on the current environmental conditions of the computing device δ, such as a `temperature` sensor—modelling construct `sense` in ScaFi;
  - *relational sensors* `r`, modelling construct `nbrvar` in ScaFi, which in addition depend also on a specific neighbour device δ' (e.g., `nbrRange`, which measures the distance with a neighbour device).

  In case `e` is a binary built-in function `b`, we shall write $\mathtt{e}_1\ \mathtt{b}\ \mathtt{e}_2$ for the function call $\mathtt{b}(\mathtt{e}_1, \mathtt{e}_2)$ whenever convenient for readability of the whole expression in which it is contained.

The key constructs of the calculus are:

- *Function call:* $\mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ is the main construct of the language. The function call evaluates to the result of applying the function value `f` produced

by the evaluation of `e` to the value of the parameters $\texttt{e}_1, \ldots, \texttt{e}_n$ *relatively to the aligned neighbours*, that is, relatively to the neighbours that in their last execution round have evaluated `e` to a function value with the same name of `f`.

- *Time evolution:* $\texttt{rep}(\texttt{e}_1)\{\texttt{e}_2\}$ is a construct for dynamically changing fields through the "repeated" application of the functional expression $\texttt{e}_2$. At the first computation round (or, more precisely, when no previous state is available—e.g., initially or at re-entrance after state was cleared out due to branching), $\texttt{e}_2$ is applied to $\texttt{e}_1$, then at each other step it is applied to the value obtained at the previous step. For instance, $\texttt{rep}(0)\{(\texttt{x}) \texttt{ => @@}\{\texttt{x} + 1\}\}$ counts how many rounds each device has computed (from the beginning, or more generally, since that piece of state was missing).
- *Neighbourhood interaction:* $\texttt{foldhood}(\texttt{e}_1)(\texttt{e}_2)\{\texttt{e}_3\}$ and $\texttt{nbr}\{\texttt{e}\}$ model device-to-device interaction. The `foldhood` construct evaluates expression $\texttt{e}_3$ against every aligned neighbour (including the device itself), then aggregates the values collected through $\texttt{e}_2$ together with the initial value $\texttt{e}_1$. The `nbr` construct tags expressions `e` signalling that (when evaluated against a neighbour) the value of `e` has to be gathered from neighbours (and not directly evaluated). Such behaviour is implemented via a conceptual broadcast of the values evaluated for `e`. Subexpressions of $\texttt{e}_3$ not containing `nbr` are *not* gathered from neighbours instead.

  As an example, consider the expression

  $$\texttt{foldhood}(2)(\texttt{+})\{\texttt{min}(\texttt{nbr}\{\texttt{temperature}()\}, \texttt{temperature}())\}$$

  evaluated in device δ_1 (in which $\texttt{temperature}() = 10$) with neighbours δ_2 and δ_3 (in which `temperature`() gave 15 and 5 in their last evaluation round, orderly). The result of the expression is then computed adding 2, $\texttt{min}(10, 10)$, $\texttt{min}(15, 10)$ and $\texttt{min}(5, 10)$ for a final value of 27.

Note that, according to the explanation given above, calling a declared or anonymous function acts as a branch, with each function in the range applied only on the subspace of devices holding a function with the same tag. In particular, we write $\texttt{branch}(\texttt{e}_1)\{\texttt{e}_2\}\{\texttt{e}_3\}$ as a shorthand for

`mux`($\mathtt{e}_1$, () => @@{$\mathtt{e}_2$}, () => @@{$\mathtt{e}_3$})()—see the discussion at the end of Section 7.2.2.

7.3.2 Typing

We now present a type system for FSCAFI. Since the type system is a customisation of the Hindley-Milner type system[5] [DM82], there is an algorithm (not presented here) that, given an expression `e` and type assumptions for its free variables, either fails (if the expression cannot be typed under the given type assumptions) or returns its *principal type*, i.e., a type such that all the types that can be assigned to `e` by the type inference rules can be obtained from the principal type by substituting type variables with types. The syntax of type and type schemes is presented in Figure 7.3 (top), where B ranges over the built-in types provided by the language (such as `num`, `bool`, `pair`(T_1, T_2), `list`(T)). The set of type variables occurring in a type T is denoted by $\mathbf{FTV}(T)$.

Type environments, ranged over by $\mathcal{A}$ and written $\overline{\mathtt{x}} : \overline{T}$, are used to collect type assumptions for program variables (i.e., formal parameters of functions). *Type-scheme environments*, ranged over by $\mathcal{D}$ and written $\overline{\mathtt{v}} : \overline{TS}$, are used to collect the type schemes for built-in constructors and built-in operators together with the type schemes inferred for user-defined functions. In particular, the distinguished *built-in type-scheme environment* $\mathcal{B}$ associates a type scheme to each built-in constructor `c` and to each built-in function `b`—Figure 7.4 shows the type schemes for the built-in constructors and built-in functions used throughout this chapter.

The typing judgement for expressions is of the form "$\mathcal{D}; \mathcal{A} \vdash \mathtt{e} : T$", to be read: "`e` has type T under the type-scheme assumptions $\mathcal{D}$ (for built-in constructors and for built-in and user-defined functions) and the type assumptions $\mathcal{A}$ (for the program variables occurring in `e`), respectively". As a standard syntax in type systems [IPW01], given $\overline{T} = T_1, \ldots, T_n$ and $\overline{\mathtt{e}} = \mathtt{e}_1, \ldots, \mathtt{e}_n$ ($n \geq 0$), we write $\mathcal{D}; \mathcal{A} \vdash \overline{\mathtt{e}} : \overline{T}$ as short for $\mathcal{D}; \mathcal{A} \vdash \mathtt{e}_1 : T_1 \cdots \mathcal{D}; \mathcal{A} \vdash \mathtt{e}_n : T_n$.

The typing rules for expressions are presented in Figure 7.3 (bottom). The rules for variables ([T-VAR]), data values ([T-DAT]), anonymous function expressions ([T-A-FUN]), built-in or defined function names ([T-N-FUN]), and function application

[5] Scala's type system is not Hindley-Milner: so, in SCAFI, while semantics is not affected, for typing we would need additional type annotations that would not be necessary for FSCAFI.

Types:

$$T ::= t \mid B \mid (\overline{T}) \to T \quad \text{type}$$
$$TS ::= \forall \overline{t}.T \quad \text{type scheme}$$

Expression typing: $\boxed{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : T}$

$$\frac{}{\mathcal{D}; \mathcal{A}, \texttt{x} : T \vdash \texttt{x} : T}\ [\text{T-VAR}] \qquad \frac{T'[\overline{t} := \overline{T}''] = (\overline{T}) \to T \qquad \mathcal{D}; \mathcal{A} \vdash \overline{\texttt{v}} : \overline{T}}{\mathcal{D}, \texttt{c} : \forall \overline{t}.T'; \mathcal{A} \vdash \texttt{c}(\overline{\texttt{v}}) : T}\ [\text{T-DAT}]$$

$$\frac{\mathcal{D};\ \mathcal{A}, \overline{\texttt{x}} : \overline{T} \vdash \texttt{e} : T}{\mathcal{D}; \mathcal{A} \vdash (\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{@@\{e\}} : (\overline{T}) \to T}\ [\text{T-A-FUN}]$$

$$\frac{\texttt{f} \text{ is a (built-in or declared) function}}{\mathcal{D}, \texttt{f} : \forall \overline{t}.T; \mathcal{A} \vdash \texttt{f} : T[\overline{t} := \overline{T}]}\ [\text{T-N-FUN}]$$

$$\frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : (\overline{T}) \to T \qquad \mathcal{D}; \mathcal{A} \vdash \overline{\texttt{e}} : \overline{T}}{\mathcal{D}; \mathcal{A} \vdash \texttt{e}(\overline{\texttt{e}}) : T}\ [\text{T-APP}]$$

$$\frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e}_1 : T \qquad \mathcal{D}; \mathcal{A} \vdash \texttt{e}_2 : (T) \to T}{\mathcal{D}; \mathcal{A} \vdash \texttt{rep}(\texttt{e}_1)\{\texttt{e}_2\} : T}\ [\text{T-REP}] \qquad \frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : T}{\mathcal{D}; \mathcal{A} \vdash \texttt{nbr}\{\texttt{e}\} : T}\ [\text{T-NBR}]$$

$$\frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e}_1 : T \quad \mathcal{D}; \mathcal{A} \vdash \texttt{e}_2 : (T, T) \to T \quad \mathcal{D}; \mathcal{A} \vdash \texttt{e}_3 : T}{\mathcal{D}; \mathcal{A} \vdash \texttt{foldhood}(\texttt{e}_1)(\texttt{e}_2)\{\texttt{e}_3\} : T}\ [\text{T-FOLD}]$$

Function typing: $\boxed{\mathcal{D} \vdash \texttt{F} : TS}$

$$\frac{\mathcal{D}, \texttt{d} : \forall \bullet.(\overline{T}) \to T; \overline{\texttt{x}} : \overline{T} \vdash \texttt{e} : T \qquad \overline{t} = \mathbf{FTV}((\overline{T}) \to T)}{\mathcal{D} \vdash \texttt{def d}(\overline{\texttt{x}}) \texttt{ = @@\{e\}} : \forall \overline{t}.(\overline{T}) \to T}\ [\text{T-FUNCTION}]$$

Program typing: $\boxed{\vdash \texttt{P} : T}$

$$\frac{\begin{array}{l}\mathcal{D}_0 = \mathcal{B} \\ \texttt{F}_i = \texttt{def d}_i(\_) \texttt{ = @@\{\_\}} \quad \mathcal{D}_{i-1} \vdash \texttt{F}_i : TS_i \quad \mathcal{D}_i = \mathcal{D}_{i-1}, \texttt{d}_i : TS_i \quad (i \in 1..n) \\ \mathcal{D}_n; \emptyset \vdash \texttt{e} : T\end{array}}{\vdash \texttt{F}_1 \cdots \texttt{F}_n\ \texttt{e} : T}\ [\text{T-PROGRAM}]$$

Figure 7.3: Type rules for expressions, function declarations, and programs.

([T-APP]), are almost standard. Rule [T-REP] (for `rep`-expressions) ensures that both the initial value $\texttt{e}_1$ and the domain and range of function $\texttt{e}_2$ have the same type, and then assigns it to $\texttt{rep}(\texttt{e}_1)\{\texttt{e}_2\}$; rule [T-NBR] (for `nbr`-expressions) assigns to $\texttt{nbr}\{\texttt{e}\}$ the same type as `e`; and rule [T-FOLD] (for `foldhood`-expressions) ensures that $\texttt{e}_1$ and $\texttt{e}_3$ have the same type T and that $\texttt{e}_2$ has type $(T, T) \to T$, and then assigns type T to $\texttt{foldhood}(\texttt{e}_1)(\texttt{e}_2)\{\texttt{e}_3\}$.

The typing rules for declared functions ([T-FUNCTION]) and programs ([T-PROGRAM]) are almost standard.

Built-in data constructors

$\mathcal{B}(\texttt{True}) = \texttt{bool}$
$\mathcal{B}(\texttt{False}) = \texttt{bool}$
$\mathcal{B}(n) = \texttt{num}$, where n is a number
$\mathcal{B}(\texttt{Pair}) = \forall t_1 t_2.(t_1, t_2) \to \texttt{pair}(t_1, t_2)$
$\mathcal{B}(\texttt{Null}) = \forall t.\texttt{list}(t)$
$\mathcal{B}(\texttt{Cons}) = \forall t.(t, \texttt{list}(t)) \to \texttt{list}(t)$

Built-in functions: pure operators

$\mathcal{B}(\texttt{pair}) = \forall t_1 t_2.t_1 \to t_2 \to \texttt{pair}(t_1, t_2)$
$\mathcal{B}(\texttt{fst}) = \forall t_1 t_2.(\texttt{pair}(t_1, t_2)) \to t_1$
$\mathcal{B}(\texttt{snd}) = \forall t_1 t_2.(\texttt{pair}(t_1, t_2)) \to t_2$
$\mathcal{B}(\texttt{cons}) = \forall t.t \to \texttt{list}(t) \to \texttt{list}(t)$
$\mathcal{B}(\texttt{head}) = \forall t.(\texttt{list}(t)) \to t$
$\mathcal{B}(\texttt{tail}) = \forall t.(\texttt{list}(t)) \to \texttt{list}(t)$
$\mathcal{B}(\texttt{=}) = \forall t.(t, t) \to \texttt{bool}$
$\mathcal{B}(\texttt{mux}) = \forall t.(\texttt{bool}, t, t) \to t$
$\mathcal{B}(\texttt{+}) = (\texttt{num}, \texttt{num}) \to \texttt{num}$
$\mathcal{B}(\texttt{and}) = (\texttt{bool}, \texttt{bool}) \to \texttt{bool}$
$\mathcal{B}(\texttt{min}) = (\texttt{num}, \texttt{num}) \to \texttt{num}$
$\mathcal{B}(\texttt{<}) = (\texttt{num}, \texttt{num}) \to \texttt{bool}$

Built-in functions: sensors

$\mathcal{B}(\texttt{temperature}) = () \to \texttt{num}$

Built-in functions: relational sensors

$\mathcal{B}(\texttt{nbrRange}) = () \to \texttt{num}$

Figure 7.4: Type schemes for the built-in value constructors and functions used in the examples.

Example 7.1 (Typing)**.** *Consider the implementation of the gradient with obstacles (as in Section 7.2.2, translated in* FScaFi*).*

```
def gradient(source, metric) = @@{ // : (bool, ()->num) -> num
  rep(PositiveInfinity){ (distance) => @@{
    mux(source) { 0.0 }{
      foldhoodPlus(PositiveInfinity)(min){ nbr{distance} + metric() }
    }
  }}
}
branch(isObstacle){ PositiveInfinity }{ gradient(isSource) }  // : num
```

The types of the gradient function and of the main expression inferred by the type

system are inserted above as comments. By rule [T-APP] and assumptions on built-in `mux`*, the type system infers that the third argument of the* `mux` *expression must be* `num`*, since the second argument is also* `num`*. It follows that* `distance` *must be of type* `num` *(rule [T-NBR]) as well and* `metric` *must be of type* $() \to$ `num` *(rule [T-APP]), from which function* `gradient` *can be inferred to have type* $(\texttt{bool}, () \to \texttt{num}) \to \texttt{num}$ *(rule [T-FUNCTION]). The overall program has then type* `num` *(rule [T-PROGRAM]).*

7.3.3 Operational semantics: device semantics

This section presents a formal semantics of device computation as happens in FScaFi, modelling the outcome one actually achieves when executing computation rounds in ScaFi. Starting from FScaFi syntax as previously described, we shall assume a fixed program `P`. We say that "device δ *fires*", to mean that the main expression of `P` is evaluated on δ.

Remark 1 (On termination of device computation)**.** *As* FScaFi *allows recursive functions, termination of a device firing is not decidable. In the rest of the chapter we assume that only terminating programs are considered.*

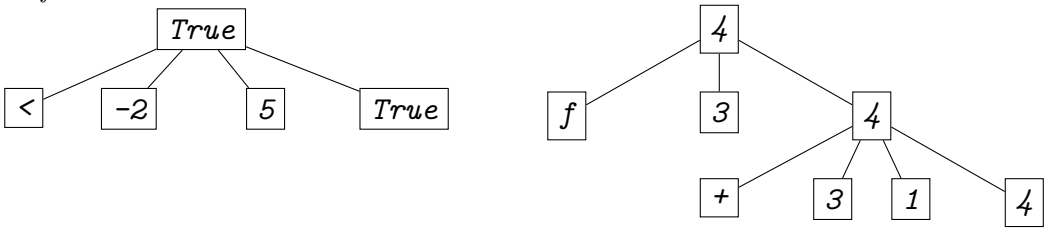
Device semantics: overall picture and preliminary definitions We model device computation by a big-step operational semantics where the result of evaluation is a *value-tree* θ (see Figure 7.5, first frame), which is an ordered tree of values, tracking the result of any evaluated subexpression. Intuitively, the evaluation of an expression at a given time in a device δ is performed against the recently-received value-trees of neighbours, namely, its outcome depends on those value-trees. The result is a new value-tree that is conversely made available to δ's neighbours (through a broadcast) for their firing; this includes δ itself, so as to support a form of state across computation rounds (note that the ScaFi implementation massively compresses the value-trees, storing only enough information for expressions to be aligned).

A *value-tree environment* Θ is a map from device identifiers to value-trees, collecting the outcome of the last evaluation on neighbours. This is written $\overline{\delta} \mapsto \overline{\theta}$ as short for $\delta_1 \mapsto \theta_1, \ldots, \delta_n \mapsto \theta_n$. The syntax of value-trees and value-tree environments is given in Figure 7.5 (first frame).

Example 7.2 (Value-trees)**.** *The graphical representation of the value-trees*

$\theta_1 = \texttt{True}\langle < \langle\rangle, -2\langle\rangle, 5\langle\rangle, \texttt{True}\langle\rangle\rangle$ *and* $\theta_2 = 4\langle f\langle\rangle, 3\langle\rangle, 4\langle +\langle\rangle, 3\langle\rangle, 1\langle\rangle, 4\langle\rangle\rangle\rangle$

is as follows:



In the following, for sake of readability, we sometimes write the value $\texttt{v}$ as shorthand for the value-tree $\texttt{v}\langle\rangle$. Following this convention, the value-tree θ_1 is shortened to $\texttt{True}\langle <, -2, 5, \texttt{True}\rangle$, and the value-tree θ_2 is shortened to $4\langle \texttt{f}, 3, 4\langle +, 3, 1, 4\rangle\rangle$.

Figure 7.5 (second frame) defines: the auxiliary functions ρ and π for extracting the root value and a subtree of a value-tree, respectively (further explanations about function π will be given later); the extension of functions ρ and π to value-tree environments; and the auxiliary functions *name*, *args* and *body* for extracting the name, formal parameters and body of a (user-defined or anonymous) function, respectively.

The computation that takes place on a single device is formalised by the big-step operational semantics rules given in Figure 7.5 (fourth frame). The derived judgements are of the form

$$\delta, \delta'; \Theta; \sigma \vdash \texttt{e} \Downarrow \theta$$

to be read "expression $\texttt{e}$ evaluates to value-tree θ on device δ with respect to the neighbour δ', value-tree environment Θ and sensor state σ", where: *(i)* δ is the identifier of the current device and δ' is either equal to δ or is one of its neighbours; *(ii)* Θ is the field of the value-trees produced by the most recent evaluation of (an expression corresponding to) $\texttt{e}$ on δ and its neighbours; *(iii)* $\texttt{e}$ is an expression; *(iv)* the value-tree θ represents the values computed for all the expressions encountered during the evaluation of $\texttt{e}$—in particular $\rho(\theta)$ is the result value of $\texttt{e}$.

The operational semantics rules are based on rather standard rules for functional languages, extended so as to be able to evaluate a subexpression $\texttt{e}'$ of $\texttt{e}$ with respect to the value-tree environment Θ' obtained from Θ by extracting the corresponding subtree (when present) in the value-trees in the range of Θ. This

process, called *alignment*, is modelled by the auxiliary function π, defined in Figure 7.5 (second frame). Function π has two different behaviours (specified by its subscript or superscript): $\pi_i(\theta)$ extracts the i-th subtree of θ, if it is present; and $\pi^{\mathtt{f}}(\theta)$ extracts the last subtree of θ, if it is present and the root of first subtree of θ is equal to $\mathtt{f}$.

When a device δ fires, its main expression $\mathtt{e}$ is evaluated with respect to δ itself. That is, by means of a judgement where $\delta' = \delta$:

$$\delta, \delta; \Theta; \sigma \vdash \mathtt{e} \Downarrow \theta$$

A key aspect of the semantics is that, if $\mathtt{e}$ is a `foldhood`-expression $\mathtt{foldhood}(\mathtt{e}_1)(\mathtt{e}_2)\{\mathtt{e}_3\}$ then its body $\mathtt{e}_3$ is first evaluated with respect to δ, and then it is evaluated again with respect to each of the devices δ' (if any) in $\mathcal{D}_\Theta \setminus \{\delta\}$. Because of alignment (see above), it might happen that a subexpression $\mathtt{e}'$ of $\mathtt{e}_3$ is evaluated by a judgement

$$\delta, \delta'; \Theta; \sigma \vdash \mathtt{e}' \Downarrow \theta \quad \text{where } \delta \neq \delta' \notin \mathcal{D}_\Theta$$

and, if the evaluation of $\mathtt{e}'$ exploits the device δ', then the evaluation of $\mathtt{e}_3$ with respect to δ' *fails* and the evaluation of the `foldhood`-expression $\mathtt{foldhood}(\mathtt{e}_1)(\mathtt{e}_2)\{\mathtt{e}_3\}$ does not consider the neighbour δ'. The evaluation rule for `foldhood`-expressions, [E-FOLD], formalises failure of evaluation with respect to a neighbour δ' by means of the auxiliary predicate

$$\delta, \delta'; \Theta; \sigma \vdash \mathtt{e} \text{ FAIL}$$

to be read "expression $\mathtt{e}$ fails to evaluate on device δ against neighbour δ' with respect to value-tree environment Θ and sensor state σ", which is formalised by the big-step operational semantics rules given in Figure 7.6.

Device semantics: rules for expression evaluation We start by explaining the rules in Figure 7.5 (fourth frame), then we will explain the rules in Figure 7.6.

Rule [E-VAL] implements the evaluation of an expression that is already a value. For instance, evaluating the expression `1` produces (by Rule [E-VAL]) the value-tree

$\theta ::= \mathtt{v}\langle\overline{\theta}\rangle$ value-tree $\qquad \Theta ::= \overline{\delta} \mapsto \overline{\theta}$ value-tree environment

Auxiliary functions and syntactic shorthands:

$name((\overline{\mathtt{x}}) \overset{\tau}{\texttt{=>}} \texttt{@@\{e\}}) = \tau \quad name(\mathtt{d}) = \mathtt{d} \quad name(\mathtt{b}) = \mathtt{b}$

$args((\overline{\mathtt{x}}) \overset{\tau}{\texttt{=>}} \texttt{@@\{e\}}) = \overline{\mathtt{x}} \quad args(\mathtt{d}) = \overline{\mathtt{x}} \quad \rho(\mathtt{v}\langle\overline{\theta}\rangle) = \mathtt{v}$

$body((\overline{\mathtt{x}}) \overset{\tau}{\texttt{=>}} \texttt{@@\{e\}}) = \mathtt{e} \quad body(\mathtt{d}) = \mathtt{e}$ (if $\texttt{def d}(\overline{\mathtt{x}})\texttt{ = @@\{e\}}$)

$\pi_i(\mathtt{v}\langle\theta_1, \ldots, \theta_n\rangle) = \theta_i$ if $1 \leq i \leq n$ else $\bullet$

$\pi^{\mathtt{f}}(\mathtt{v}\langle\theta_1, \ldots, \theta_{n+2}\rangle) = \theta_{n+2}$ if $name(\rho(\theta_1)) = name(\mathtt{f})$ else $\bullet$

For $aux \in \rho, \pi_i, \pi^{\mathtt{f}}$:
$$\begin{cases} aux(\bullet) = \bullet \\ aux(\delta \mapsto \theta, \Theta) = aux(\Theta) & \text{if } aux(\theta) = \bullet \\ aux(\delta \mapsto \theta, \Theta) = \delta \mapsto aux(\theta), aux(\Theta) & \text{if } aux(\theta) \neq \bullet \end{cases}$$

$\delta, \delta'; \overline{\pi}(\Theta); \sigma \vdash \overline{\mathtt{e}} \Downarrow \overline{\theta}$ where $|\overline{\mathtt{e}}| = n$ for $\delta, \delta'; \pi_i(\Theta); \sigma \vdash \mathtt{e}_i \Downarrow \theta_i,\ i = 1, \ldots, n$

$\rho(\overline{\theta})$ where $|\overline{\theta}| = n$ for $\rho(\theta_1), \ldots, \rho(\theta_n)$

$\overline{\mathtt{x}} := \rho(\overline{\theta})$ where $|\overline{\mathtt{x}}| = n$ for $\mathtt{x}_1 := \rho(\theta_1)\ \ldots\ \mathtt{x}_n := \rho(\theta_n)$

Rules for expression evaluation: $\boxed{\delta, \delta'; \Theta; \sigma \vdash \mathtt{e} \Downarrow \theta}$

[E-VAL]
$$\frac{}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{v} \Downarrow \mathtt{v}\langle\rangle}$$

[E-B-APP]
$$\frac{\begin{array}{l}\delta, \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e} \Downarrow \theta \quad \delta, \delta'; \pi_{i+1}(\Theta); \sigma \vdash \mathtt{e}_i \Downarrow \theta_i \quad \forall i \in 1, \ldots, n \\ \mathtt{v} = (\!|\mathtt{b}|\!)^{\pi^{\mathtt{b}}(\Theta), \sigma}_{\delta, \delta'}(\rho(\overline{\theta})) \quad \mathtt{b} = \rho(\theta) \text{ not rel.} \wedge \delta = \delta' \wedge \delta' \in \mathcal{D}_{\pi^{\mathtt{b}}(\Theta)}\end{array}}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{e}(\overline{\mathtt{e}}) \Downarrow \mathtt{v}\langle\theta, \overline{\theta}, \mathtt{v}\rangle}$$

[E-D-APP]
$$\frac{\begin{array}{l}\delta, \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e} \Downarrow \theta \quad \delta, \delta'; \pi_{i+1}(\Theta); \sigma \vdash \mathtt{e}_i \Downarrow \theta_i \quad \forall i \in 1, \ldots, n \\ \mathtt{f} = \rho(\theta) \text{ not built-in} \quad \delta, \delta'; \pi^{\mathtt{f}}(\Theta); \sigma \vdash body(\mathtt{f})[args(\mathtt{f}) := \rho(\overline{\theta})] \Downarrow \theta'\end{array}}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{e}(\overline{\mathtt{e}}) \Downarrow \rho(\theta')\langle\theta, \overline{\theta}, \theta'\rangle}$$

[E-REP]
$$\frac{\begin{array}{ll}\delta, \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta_1 & \mathtt{v}_1 = \rho(\theta_1) \\ \delta, \delta; \pi_2(\Theta); \sigma \vdash \mathtt{e}_2(\mathtt{v}_0) \Downarrow \theta_2 & \mathtt{v}_2 = \rho(\theta_2)\end{array} \quad \mathtt{v}_0 = \begin{cases} \rho(\pi_2(\Theta))(\delta) & \delta \in \mathcal{D}_\Theta \\ \mathtt{v}_1 & \text{or else} \end{cases}}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{rep}(\mathtt{e}_1)\{\mathtt{e}_2\} \Downarrow \mathtt{v}_2\langle\theta_1, \theta_2\rangle}$$

[E-NBR]
$$\frac{\delta \neq \delta' \in \mathcal{D}_\Theta \qquad \theta = \Theta(\delta')}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{nbr}\{\mathtt{e}\} \Downarrow \theta}$$

[E-NBR-LOC]
$$\frac{\delta, \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e} \Downarrow \theta}{\delta, \delta; \Theta; \sigma \vdash \mathtt{nbr}\{\mathtt{e}\} \Downarrow \rho(\theta)\langle\theta\rangle}$$

[E-FOLD]
$$\frac{\begin{array}{ll}\delta, \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta^1 & \delta, \delta; \pi_2(\Theta); \sigma \vdash \mathtt{e}_2 \Downarrow \theta_0\ \mathtt{f} = \rho(\theta_0) \\ \delta_1, \ldots \delta_n = \mathcal{D}_\Theta \cup \{\delta\} & n \geq m \geq 1 \qquad \delta_1 = \delta \\ \delta, \delta_i; \pi_3(\Theta); \sigma \vdash \mathtt{e}_3 \Downarrow \theta_i & \forall i \in 1, \ldots, m \\ \delta, \delta_j; \pi_3(\Theta); \sigma \vdash \mathtt{e}_3 \text{ FAIL} & \text{for all } j \in m+1, \ldots, n \\ \delta, \delta; \emptyset; \sigma \vdash \mathtt{f}(\rho(\theta^i), \rho(\theta_i)) \Downarrow \theta^{i+1} & \text{for all } i \in 1, \ldots, m\end{array}}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{foldhood}(\mathtt{e}_1)(\mathtt{e}_2)\{\mathtt{e}_3\} \Downarrow \rho(\theta^{m+1})\langle\theta^1, \theta_0, \theta_1\rangle}$$

Figure 7.5: Big-step operational semantics for expression evaluation.

Auxiliary rules for expression evaluation failure: $\boxed{\delta, \delta'; \Theta; \sigma \vdash \mathtt{e} \text{ FAIL}}$

$$\text{[E-NBR-FAIL]}\ \frac{\delta \neq \delta' \notin \mathcal{D}_\Theta}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{nbr\{e\}} \text{ FAIL}}$$

$$\text{[E-R-APP-FAIL]}\ \frac{\begin{array}{ll}\delta, \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e} \Downarrow \theta & \delta, \delta'; \pi_{i+1}(\Theta); \sigma \vdash \mathtt{e}_i \Downarrow \theta,\ \forall\, i \in 1, ..., n \\ \mathtt{r} = \rho(\theta) \text{ is a relational built-in} & \delta \neq \delta' \notin \mathcal{D}_{\pi^{\mathtt{r}}(\Theta)}\end{array}}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{e}(\overline{\mathtt{e}}) \text{ FAIL}}$$

$$\text{[E-APP-ARG-FAIL]}\ \frac{\begin{array}{lll}\delta, \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e} \Downarrow \theta & & \overline{\mathtt{e}} = \mathtt{e}_1, ..., \mathtt{e}_n \qquad n \geq 0 \\ \delta, \delta'; \pi_{i+1}(\Theta); \sigma \vdash \mathtt{e}_i \Downarrow \theta_i & & \text{for all } i \in 1, ..., m < n \\ \delta, \delta'; \pi_{m+2}(\Theta); \sigma \vdash \mathtt{e}_{m+1} \text{ FAIL} & & \end{array}}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{e}(\overline{\mathtt{e}}) \text{ FAIL}}$$

$$\text{[E-D-APP-FAIL]}\ \frac{\begin{array}{ll}\delta, \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e} \Downarrow \theta & \delta, \delta'; \pi_{i+1}(\Theta); \sigma \vdash \mathtt{e}_i \Downarrow \theta_i, \forall i \in 1, ..., n \\ \mathtt{f} = \rho(\theta) \text{ not built-in} & \\ \delta, \delta'; \pi^{\mathtt{f}}(\Theta); \sigma \vdash \mathit{body}(\mathtt{f})[\mathit{args}(\mathtt{f}) := \rho(\overline{\theta})] \text{ FAIL} & \end{array}}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{e}(\overline{\mathtt{e}}) \text{ FAIL}}$$

Figure 7.6: Big-step operational semantics for expression evaluation (auxiliary rules for expression evaluation failure).

$1\langle\rangle$, while evaluating the expression + produces the value-tree $+\langle\rangle$.

Rules [E-B-APP] and [E-D-APP] model function application $\mathtt{e}(\mathtt{e}_1 \cdots \mathtt{e}_n)$. In case $\mathtt{e}$ evaluates to a built-in function $\mathtt{b}$, rule [E-B-APP] is used, whose behaviour is driven by the special auxiliary function $(\!|\mathtt{b}|\!)^{\Theta,\sigma}_{\delta,\delta'}$ (operational interpretation of $\mathtt{b}$), whose actual definition is abstracted away.

Example 7.3 (Built-in function application)**.** *Evaluating the expression* $<(-2, 5)$ *produces the value-tree* $\theta_1 = \mathtt{True}\langle <, -2, 5, \mathtt{True}\rangle$ *introduced in Example 7.2. The operational interpretation* $(\!|<|\!)^{\Theta,\sigma}_{\delta,\delta'}$ *of* $<$ *is the following (notice that this interpretation does not depend on* $\Theta, \sigma, \delta, \delta'$*, since* $<$ *is a pure mathematical operator):*

$$(\!|<|\!)^{\Theta,\sigma}_{\delta,\delta'} = \lambda x.\lambda y. \begin{cases} \mathtt{True} & x < y \\ \mathtt{False} & \mathit{otherwise} \end{cases}$$

The value of the whole expression, `True` *(the root of the last subtree of the value-tree), has been computed by using rule [E-B-APP] to evaluate the application* $(\!|<$

$\rangle\!\rangle_{\delta,\delta'}^{\Theta,\sigma}(-2,5)$ *of the less-then operator* $<$ *(the root of the first subtree of the value-tree) to the values* -2 *(the root of the second subtree of the value-tree) and* 5 *(the root of the third subtree of the value-tree).*

In case `e` evaluates to a user-defined or anonymous function `f`, rule [E-D-APP] is used: it performs domain restriction $\pi^{\mathtt{f}}(\Theta)$ (thus discarding devices that did not apply the same function `f`, for which no consistent information on the application of `f` is present), then continues the evaluation by substituting the arguments into the body of `f`. We remark that we do not assume that Θ is empty whenever it does not contain δ. In fact, in any round where `e` evaluates to a function `f` for the first time on a device, $\mathtt{f}(\overline{\mathtt{e}})$ will be evaluated with respect to an environment not containing δ but possibly containing other devices (whose `e` evaluated to `f` in their previous round of computation).

Example 7.4 (Defined or anonymous function application)**.** *Evaluating the expression* $\mathtt{f}(3)$*, where* `f` *is the name of the declared function* `def f(x) = @@{x + 1}`*, produces the value-tree* $\theta_2 = 4\langle \mathit{f}, 3, 4\langle +, 3, 1, 4\rangle\rangle$ *introduced in Example 7.2. The value of the whole expression,* 4 *(the root of* θ_2*), which has been computed by using rule [E-D-APP], is the root of the last subtree of* θ_2*, which is produced by the evaluation of the expression* $3 + 1$ *(obtained from the body of* `f` *by replacing* `x` *with* 3*). Evaluating the similar expression* $\mathtt{f}(3)$ *where* `f` *is the anonymous function* $((\mathtt{x}) \overset{\tau}{\mathtt{=>}} \mathtt{@@}\{\mathtt{x} + 1\})$*, produces the same value-tree* θ_2 *by the same rule [E-D-APP].*

Rule [E-REP] implements internal state evolution through computational rounds: on the first firing of a device, $\mathtt{rep}(\mathtt{e}_1)\{\mathtt{e}_2\}$ evaluates to $\mathtt{e}_2(\mathtt{e}_1)$, then it evaluates to $\mathtt{e}_2(\mathtt{v})$ where `v` is the value calculated in the previous round.

Example 7.5 (Time evolution)**.** *To illustrate rule [E-REP], as well as computational rounds, we consider the program* $\mathtt{rep}(3)\{\mathtt{f}\}$ *where* `f` *is the anonymous function* $(\mathtt{x}) \overset{\tau}{\mathtt{=>}} \mathtt{@@}\{\mathtt{x} + 1\}$ *introduced in Example 7.4. The first firing of a device* δ *is performed against the empty tree environment. Therefore, according to rule [E-REP], evaluating* $\mathtt{rep}(3)\{\mathtt{f}\}$ *produces the value-tree* $\theta = 4\langle 3, \theta_2\rangle$ *where* $\theta_2 = 4\langle \mathit{f}, 3, 4\langle +, 3, 1, 4\rangle\rangle$ *is the value-tree (introduced in Example 7.2) produced by evaluating the expression* $\mathtt{f}(3)$ *as described in Example 7.4. The overall result of the firing is the root* 4 *of* θ*. Any subsequent firing of the device* δ *is performed with respect to a value-tree environment* Θ *that associates to* δ *the outcome* θ *of*

the most recent firing of δ. Therefore, evaluating `rep(3){f}` *at the second firing produces the value-tree $\theta' = 5\langle 4, \theta'_2\rangle$ where $\theta'_2 = 5\langle \mathtt{f}, 4, 5\langle +, 4, 1, 5\rangle\rangle$ is the value-tree produced by evaluating the expression* `f(4)`*, where 4 is the root of θ. Hence, the results of the firings are 4, 5, 6, and so on.*

Rules [E-NBR] and [E-NBR-LOC] model device interaction (together with [E-FOLD] which we shall consider later). When an `nbr`-expression is not evaluated against a neighbour (that is, $\delta' = \delta$), by Rule [E-NBR-LOC] the `nbr` operator is discarded and the evaluation continues. Whenever instead an `nbr`-expression is evaluated against a neighbour (that is, $\delta' \neq \delta$), by Rule [E-NBR] the expression directly evaluates to $\Theta(\delta')$ (which is the value-tree calculated by device δ' in its last computational round). Notice that it could be possible that δ' is not in the domain of Θ due to alignment operations performed in subexpressions of the enclosing instance of `foldhood`. In this case, no rule is applicable and the `nbr`-expression *fails*, causing δ' to be ignored by the enclosing `foldhood` operator (see Rule [E-FOLD]).

Rule [E-FOLD] implements collection and aggregation of results from neighbours, proceeding in the following steps:

- Evaluate the initial value $\mathtt{e}_1$ with respect to the current device obtaining the value-tree θ^1.
- Evaluate the aggregator $\mathtt{e}_2$ with respect to the current device obtaining θ_0 with root `f`.
- Evaluate the body $\mathtt{e}_3$ with respect to the current device $\delta = \delta_1$ and with respect to every neighbour $\delta' \in \mathcal{D}_{\pi_3(\Theta)} \setminus \{\delta\}$ and *consider* only the $m \geq 1$ neighbours $\delta_1, ..., \delta_m$ *for which the evaluation does not fail*, obtaining the value-trees $\theta_1, ..., \theta_m$, respectively.[6]
- Aggregate the values $\rho(\theta_i)$ $(1 \leq i \leq m)$ computed above together with the initial value $\rho(\theta^1)$ via function `f`, obtaining the final outcome $\rho(\theta^{m+1})$. Such aggregation is performed with respect to the current device and the empty environment, since the value-trees of the aggregation process cannot be easily related with one another (and thus are not stored in the final outcome of

[6] If the aggregator `f` is associative and commutative, then the result of the aggregation does not depend on the order in which the neighbours $\delta' \in \mathcal{D}_{\pi_3(\Theta)} \setminus \{\delta\}$ are considered. To ensure determinism even in the case when the aggregator `f` is not associative and commutative, we assume that the neighbours are considered according to a given total order on device identifiers.

the computation). In other words, the aggregator `f` is forced to be a "pure" function independent of the current device and environment (even though the expression $\mathtt{e}_2$ as a whole might depend on the environment).

Failure of evaluation against a neighbour is formalised by means of the auxiliary judgement $\delta, \delta'; \Theta; \sigma \vdash \mathtt{e}$ FAIL defined by the rules in Figure 7.6. Rules [E-NBR-FAIL] and [E-R-APP-FAIL] model the failure sources, while the other rules model failure propagation.

The values aggregated by `foldhood` include the value of $\mathtt{e}_3$ in the current device δ. However, an exclusive folding operation `foldhoodPlus` that only considers neighbours and not the device itself can be easily encoded on top of it (see Section 7.2.2) and is in fact available in ScaFi.

Example 7.6 (Neighbourhood interaction)**.** *To illustrate rules [E-FOLD], [E-NBR] and [E-NBR-LOC], we consider program*

$$\mathtt{foldhood}(2)(\mathtt{+})\{\ \mathtt{min}(\mathtt{nbr}\{\mathtt{temperature}()\}, \mathtt{temperature}())\ \}$$

evaluated in device δ_1 (in which $\mathtt{temperature}() = 10$) with neighbours δ_2 (in which $\mathtt{temperature}() = 15$) and δ_3 (in which $\mathtt{temperature}() = 5$). By Rule [E-FOLD], the first two subexpressions of the `foldhood`*-expression are evaluated with respect to δ_1 into the value-trees $\theta^1, \theta_0, \theta_1$ which will constitute the branches of the final tree. They are $\theta^1 = 2\langle\rangle$ and $\theta_0 = \mathtt{+}\langle\rangle$, each of them obtained by Rule [E-VAL]. Then, the third subexpression is evaluated against δ_1, δ_2 and δ_3, obtaining:*

$$\theta_1 = 10\langle \mathtt{min}, 10\langle 10\langle \mathtt{temperature}, 10\rangle\rangle, 10\langle \mathtt{temperature}, 10\rangle, 10\rangle,$$
$$\theta_2 = 10\langle \mathtt{min}, 15\langle 15\langle \mathtt{temperature}, 15\rangle\rangle, 10\langle \mathtt{temperature}, 10\rangle, 10\rangle,$$
$$\theta_3 = 5\langle \mathtt{min}, 5\langle 5\langle \mathtt{temperature}, 5\rangle\rangle, 10\langle \mathtt{temperature}, 10\rangle, 5\rangle$$

the first one (θ_1) obtained through three applications of Rule [E-B-APP] and one of Rule [E-NBR-LOC], and the other two (θ_2 and θ_3) obtained through three applications of Rule [E-B-APP] and one of Rule [E-NBR]. The roots of these value-trees are then combined through operator `+`*, together with the initial value 2, for a total result of $2 + 10 + 10 + 5 = 27$ which is the root of the final value-tree $27\langle 2, \mathtt{+}, \theta_1\rangle$.*

Rules [E-VAL], [E-REP], [E-FOLD] are independent of the neighbour δ' against which

the expression is computed (since δ' does not occur in the premises of those rules). Rules [E-B-APP] and [E-D-APP] simply pass δ' through, allowing subexpressions to make use of it (including evaluation of built-in relational sensors `r`). The neighbour device δ' is then non-trivially used only in rules [E-NBR], [E-NBR-LOC], [E-NBR-FAIL] and [E-R-APP-FAIL].

We say that a neighbour is *considered* by the evaluation of a `foldhood`-expression to mean that it contributes to the result of the expression. Because of the interplay between neighbourhood interaction and branching (i.e., function call) only a subset of the neighbourhood of a device might be considered by a `foldhood`-expression.

Example 7.7 (Neighbourhood interaction and branching)**.** *In order to illustrate the alignment process, guiding neighbour interaction through branching statements, consider the* gradient with obstacles *function discussed in Example 7.1.*

```
def gradient(source, metric) = @@{
  rep(PositiveInfinity){ (distance) => @@{
    mux( source, 0.0, foldhoodPlus(PositiveInfinity)(min){ nbr{distance}
     + metric() } )
} } }
branch(isObstacle){ PositiveInfinity }{ gradient(isSource) }
```

Expanding the syntactic sugar, the `branch` *statement corresponds to the execution of a different anonymous function depending on the value of* `isObstacle`*:*

```
mux( isObstacle, () => @@{PositiveInfinity}, () => @@{gradient(
    isSource)} )()
```

Assume that device δ_0 evaluates this program with respect to $\Theta = \{\delta_0 \mapsto \theta_0, \delta_1 \mapsto \theta_1, \delta_2 \mapsto \theta_2\}$, where `isObstacle` *is true in δ_2 and false on the other devices. Thus, the execution of the* `mux` *statement produces* $\mathtt{f}_\bot = ()$ `=> @@{gradient(isSource)}` *on δ_0 and δ_1, while it produces* $\mathtt{f}_\top = ()$ `=> @@{PositiveInfinity}` *on δ_2.*

The evaluation of the main expression is performed through rule [E-D-APP]. First, the function to be applied is computed as the result of the `mux` *expression. Then, the body* `gradient(isSource)` *is computed with respect to the environment $\pi^{\mathtt{f}_\bot}(\Theta) = \{\delta_0 \mapsto \pi_2(\theta_0), \delta_1 \mapsto \pi_2(\theta_1)\}$: the value-tree of device δ_2 is removed since it corresponded to the evaluation of $\mathtt{f}_\top$. The evaluation of* `gradient(isSource)` *will then require the evaluation of the* `foldhoodPlus` *expression, in which only*

devices δ_0 and δ_1 will be considered (since δ_2 has already been discarded).

Implementation and performance considerations The implementation of the device semantics in ScaFi is structured to manage the population of a value tree upon evaluation of construct occurrences while respecting the rules specified in Figure 7.5. In doing so, naive implementations for the constructs may lead to performance issues. For instance, a sequence of nested `foldhood`-operators (not interleaved by `nbr`-operators) can lead to an exponential evaluation time, as, for each aligned neighbour, an expression which in turn depends on all aligned neighbours would need to be evaluated. To solve the issue, in ScaFi the outcome of `foldhood` and `rep` subexpressions is "memoised" in order to prevent subsequent re-evaluation (since such expressions are independent of the neighbour against which are evaluated). This addresses the performance issues of nested `foldhood`-operators, thus ensuring linear evaluation time.

7.3.4 Operational semantics: network semantics

We now provide an operational semantics for the evolution of whole networks, namely, for modelling the distributed evolution of computational fields over time. Figure 7.7 (top) defines key syntactic elements to this end:

- Ψ is a computational field that models the overall state as a map from device identifiers to value-tree environments. From it, we can define the field ϕ summarising the current status of the network as the map from device identifiers to value-trees: $\phi(\delta) = \Psi(\delta)(\delta)$.
- τ models *network topology*, namely, a directed neighbouring graph, as a map from device identifiers to a set of identifiers.
- Σ models *sensor (distributed) state*, as a map from device identifiers to (local) sensors (i.e., sensor name/value maps).
- *Env* (a pair of topology and sensor state) models the network environment.
- N (a pair of a field and environment) models a whole network configuration.

We use the following notation for computational fields. Let $\overline{\delta} \mapsto \Theta$ denote the map sending each device identifier in $\overline{\delta}$ to the same value-tree environment Θ.

Network configurations and action labels:

Ψ	$::=$	$\overline{\delta} \mapsto \overline{\Theta}$	computational field
τ	$::=$	$\overline{\delta} \mapsto \overline{I}$	topology
Σ	$::=$	$\overline{\delta} \mapsto \overline{\sigma}$	sensors-map
Env	$::=$	τ, Σ	environment
N	$::=$	$\langle Env; \Psi \rangle$	network configuration
act	$::=$	$\delta \mid env$	action label

Environment well-formedness:

$WFE(\tau, \Sigma)$ holds if τ, Σ have same domain, and τ's values do not escape it.

Transition rules for network evolution: $\boxed{N \xrightarrow{act} N}$

$$\text{[N-FIR]}\quad \frac{\delta, \delta; F(\Psi)(\delta); \Sigma(\delta) \vdash \mathtt{e_{main}} \Downarrow \theta \quad \Psi_1 = \overline{\delta} \mapsto \{\delta \mapsto \theta\} \qquad Env = \tau, \Sigma \quad \tau(\delta) = \overline{\delta}}{\langle Env; \Psi \rangle \xrightarrow{\delta} \langle Env; F(\Psi)[\Psi_1] \rangle}$$

$$\text{[N-ENV]}\quad \frac{WFE(Env') \qquad Env' = \tau, \overline{\delta} \mapsto \overline{\sigma} \qquad \Psi_0 = \overline{\delta} \mapsto \emptyset}{\langle Env; \Psi \rangle \xrightarrow{env} \langle Env'; \Psi_0[\Psi] \rangle}$$

Figure 7.7: Small-step operational semantics for network evolution.

Let $\Theta_0[\Theta_1]$ denote the value-tree environment with domain $\mathcal{D}_{\Theta_0} \cup \mathcal{D}_{\Theta_1}$ coinciding with Θ_1 in the domain of Θ_1 and with Θ_0 otherwise. Let $\Psi_0[\Psi_1]$ denote the computational field with the *same domain* as Ψ_0 made of $\delta \mapsto \Psi_0(\delta)[\Psi_1(\delta)]$ for all δ in the domain of Ψ_1, $\delta \mapsto \Psi_0(\delta)$ otherwise. The notation $F_\delta(\cdot)$ used in rule [N-FIR], Figure 7.7 (bottom), models a filtering operation that clears out old stored value-trees from $\Psi(\delta)$, implicitly based on space/time tags.[7]

We define network operational semantics in terms of small-steps transitions of the kind $N \xrightarrow{act} N'$, where act is either a device identifier in case it represents its firing, or label env to model any environment change. This is formalised in Figure 7.7 (bottom).

Rule [N-FIR] models a computation round (firing) at device δ: it takes the local value-tree environment filtered out of old values $F_\delta(\Psi)(\delta)$; then, by the single device semantics, it obtains the device's value-tree θ, which is used to update the network configuration of δ's neighbours—the local sensors $\Sigma(\delta)$ are used by

[7] For example, the filter may remove value-trees that were stored before $t - \Delta t$, where t is the time of the current firing and Δt is a decay parameter of the filter.

the auxiliary function $(\!|\mathtt{b}|\!)^{\Theta,\Sigma(\delta)}_{\delta,\delta'}$ that gives the semantics to the built-in functions. Notice that expression $\mathtt{e_{main}}$ is always evaluated in the device itself (that is, against no neighbour).

Rule [N-ENV] models environment change to a new well-formed environment Env'. Let $\overline{\delta}$ be the domain of Env'. We first construct a field Ψ_0 associating to all the devices of Env' the empty context $\emptyset$. Then, we adapt the existing field Ψ to the new set of devices: $\Psi_0[\Psi]$ automatically handles removal of devices, mapping of new devices to the empty context, and retention of existing contexts in the other devices.

Example 7.8 (Network evolution)**.** *Consider the program in Example 7.6:*

$$\texttt{foldhood}(2)(+)\{\ \mathit{min}(\texttt{nbr}\{\mathit{temperature}()\}, \mathit{temperature}())\ \}$$

and let $\theta^n = n\langle \mathit{min}, n\langle n\langle \texttt{temperature}, n\rangle\rangle, n\langle \texttt{temperature}, n\rangle, n\rangle$ *be the result of evaluation of* $\mathit{min}(\texttt{nbr}\{\mathit{temperature}()\}, \mathit{temperature}())$ *in a device where* $\mathit{temperature}() = n$*.*

We start from a configuration $N_0 = \langle \tau, \Sigma; \Psi_0\rangle$ *with three devices* $\overline{\delta}$*, so that* $\tau = \overline{\delta} \mapsto \{\delta_1, \delta_2, \delta_3\}$ *(all devices are connected),* $\Psi_0 = \overline{\delta} \mapsto \emptyset$ *(devices do not hold any information) and*

$$\Sigma = \delta_1 \mapsto \{t = 10\}, \delta_2 \mapsto \{t = 15\}, \delta_3 \mapsto \{t = 5\}$$

(temperatures are as in Example 7.6).

After transition $N_0 \xrightarrow{\delta_2} N_1$*, the computational field* Ψ_0 *is updated by sending the result* $\theta_0 = 17\langle 2, +, \theta^{15}\rangle$ *of the computation of* δ_2 *(with respect to its empty environment) to every device, obtaining* $\Psi_1 = \overline{\delta} \mapsto \{\delta_2 \mapsto \theta_0\}$*. Then, another transition takes place:* $N_1 \xrightarrow{\delta_3} N_2$*, where* Ψ_1 *is further updated with the result* $\theta_1 = 12\langle 2, +, \theta^5\rangle$ *of the computation of* δ_3 *(with respect to the information received from* δ_2*), obtaining* $\Psi_2 = \overline{\delta} \mapsto \{\delta_2 \mapsto \theta_0, \delta_3 \mapsto \theta_1\}$*. Finally, transition* $N_2 \xrightarrow{\delta_1} N_3$ *happens as described in Example 7.6, producing* $\Psi_3 = \overline{\delta} \mapsto \{\delta_1 \mapsto \theta_2, \delta_2 \mapsto \theta_0, \delta_3 \mapsto \theta_1\}$ *where* $\theta_2 = 27\langle 2, +, \theta^{10}\rangle$*.*

Lastly, a transition $N_3 \xrightarrow{env} N_4$ *may happen, lowering temperatures, deleting device* δ_2*, inserting device* δ_4*, and disconnecting device* δ_1 *from device* δ_3*. The*

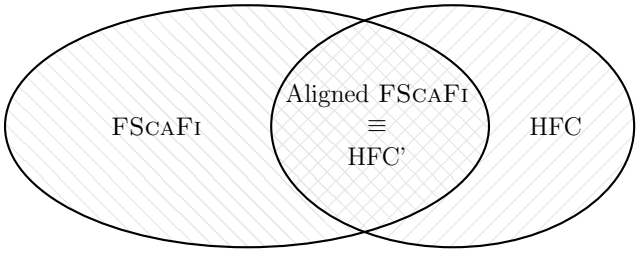


Figure 7.8: Relationship between FScaFi, HFC, and their fragments.

result is configuration $N_4 = \langle \tau', \Sigma'; \Psi_4 \rangle$ *where:*

$$\tau' = \delta_1 \mapsto \{\delta_4\}, \delta_3 \mapsto \{\delta_4\}, \delta_4 \mapsto \{\delta_1, \delta_3\}$$
$$\Sigma' = \delta_1 \mapsto \{t = 9\}, \delta_1 \mapsto \{t = 4\}, \delta_4 \mapsto \{t = 1\}$$
$$\Psi_4 = \delta_1 \mapsto \{\delta_1 \mapsto \theta_2, \delta_2 \mapsto \theta_0, \delta_3 \mapsto \theta_1\}, \delta_3 \mapsto \{\delta_1 \mapsto \theta_2, \delta_2 \mapsto \theta_0, \delta_3 \mapsto \theta_1\}, \delta_4 \mapsto \emptyset.$$

Notice that devices δ_1, δ_3 *are not aware yet of the disappearance of* δ_2*, nor of their disconnection. When one of them will fire, the filter* $F_{\delta_i}(\cdot)$ *will be able to remove the obsolete values from the corresponding value-tree environments.*

7.4 Properties and Relation with HFC

In this section, we present properties of FScaFi and relationship with the HFC minimal core calculus for Aggregate Computing [Aud+19], used as main related approach to formalise field computations.

As a preliminary result, we prove in Section 7.4.1 the device computation type preservation property for FScaFi. Then, in Section 7.4.2, we define a fragment of FScaFi, which we call *Aligned* FScaFi, where the programmer can better control the scope of operations working on neighbours: essentially, Aligned FScaFi corresponds to a fragment of HFC (that we correspondingly call HFC′) obtained by imposing restrictions on how values from neighbours can be aggregated. As a result, a formal translation is provided to make the two fragments match—see Figure 7.8.

Finally, in Section 7.4.3, we show that these fragments attain maximal expressiveness according to the definition in [Aud+18], and contain most useful

programs—e.g., self-stabilising building blocks as of [Vir+18]. Overall, these results show that ScaFi provides a different "flavour" of field computation with respect to FC (and Protelis [PVB15]), though without losing practical expressiveness.

7.4.1 Type Preservation in FScaFi

We now prove that the evaluation rules for FScaFi are deterministic and preserve types, provided that the value-tree environment used for the evaluation is coherent with the expression being evaluated according to the following definition.

Definition 1 (Well Formed Value Tree)**.** *Given a closed expression* $\mathtt{e}$*, a local-type-scheme environment* $\mathcal{D}$*, a type environment* $\mathcal{A} = \overline{\mathtt{x}} : \overline{T}$*, and a type* T *such that* $\mathcal{D}; \mathcal{A} \vdash \mathtt{e} : T$ *holds, the set* $\mathcal{C}^T_{\mathcal{D};\mathcal{A}}[\mathtt{e}]$ *of the* well-formed value-trees *for* $\mathtt{e}$ *is inductively defined as follows.* $\theta \in \mathcal{C}^T_{\mathcal{D};\mathcal{A}}[\mathtt{e}]$ *if and only if* $\mathtt{v} = \rho(\theta)$ *has type* T *(i.e.* $\mathcal{D}; \emptyset \vdash \mathtt{v} : T$*) and*

- *if* $\mathtt{e}$ *is a value,* θ *is of the form* $\mathtt{v}\langle\rangle$*;*
- *if* $\mathtt{e} = \mathtt{nbr}\{\mathtt{e}_1\}$*,* θ *is of the form* $\mathtt{v}\langle\theta_1\rangle$ *where* $\theta_1 \in \mathcal{C}^T_{\mathcal{D};\mathcal{A}}[\mathtt{e}_1]$*;*
- *if* $\mathtt{e} = \mathtt{rep}(\mathtt{e}_1)\{\mathtt{e}_2\}$*,* θ *is of the form* $\mathtt{v}\langle\theta_1, \theta_2\rangle$ *where* $\theta_1 \in \mathcal{C}^T_{\mathcal{D};\mathcal{A}}[\mathtt{e}_1]$ *and* $\theta_2 \in \mathcal{C}^T_{\mathcal{D};\mathcal{A},\mathtt{x}:T}[\mathtt{e}_2(\mathtt{x})]$*;*
- *If* $\mathtt{e} = \mathtt{foldhood}(\mathtt{e}_1)(\mathtt{e}_2)\{\mathtt{e}_3\}$*,* θ *is of the form* $\mathtt{v}\langle\theta_1, \theta_2, \theta_3\rangle$ *where* $\overline{\theta} \in \mathcal{C}^{T,(T,T)\to T,T}_{\mathcal{D};\mathcal{A}}[\overline{\mathtt{e}}]$*;*
- *if* $\mathtt{e} = \mathtt{e}'(\overline{\mathtt{e}})$ *and* $\mathcal{D}; \mathcal{A} \vdash \mathtt{e}' : T'$*,* $\mathcal{D}; \mathcal{A} \vdash \overline{\mathtt{e}} : \overline{T}$*, then* θ *is of the form* $\mathtt{v}\langle\theta', \overline{\theta}, \theta''\rangle$ *where* $\theta' \in \mathcal{C}^{T'}_{\mathcal{D};\mathcal{A}}[\mathtt{e}']$*,* $\overline{\theta} \in \mathcal{C}^{\overline{T}}_{\mathcal{D};\mathcal{A}}[\overline{\mathtt{e}}]$*, and either:*
  - $\mathtt{f} = \rho(\theta')$ *is a built-in function and* $\theta'' = \mathtt{v}\langle\rangle$*,*
  - $\mathtt{f}$ *is not a built-in function and* $\theta'' \in \mathcal{C}^T_{\mathcal{D};\mathcal{A},args(\mathtt{f}):\overline{T}}[body(\mathtt{f})]$*.*

Similarly, the set of well-formed value-tree environments *WFVTE*$(\mathcal{D}; \mathcal{A}; \mathtt{e})$ *is the set of* $\Theta = \overline{\delta} \mapsto \overline{\theta}$ *such that* $\overline{\theta} \in \mathcal{C}^T_{\mathcal{D};\mathcal{A}}[\mathtt{e}]$*.*

In other words, the above definition demands value-trees to be plausible outcomes of the evaluation of $\mathtt{e}$.

Lemma 1 (Computation Determinism)**.** *Let* $\mathtt{e}$ *be a well-typed closed expression and* $\Theta \in \mathit{WFVTE}(\mathcal{D}; \mathcal{A}; \mathtt{e})$*. Then for all device identifiers* δ*,* δ' *and sensor state* σ*:*

1. $\delta, \delta; \Theta; \sigma \vdash \mathtt{e}$ FAIL *cannot hold.*
2. *There is at most one derivation of the kind* $\delta, \delta'; \Theta; \sigma \vdash \mathtt{e} \Downarrow \theta$ *or* $\delta, \delta'; \Theta; \sigma \vdash \mathtt{e}$ FAIL*.*

Proof. See [Cas+20]. □

By this lemma, evaluation does not result on FAIL when it is performed relative to the current device (as it is the case for main expressions), and rules are deterministic. Furthermore, the evaluation rules respect the types given in Figure 7.3, provided that the *built-in interpretations respect the given types*. Formally, given $\mathtt{b}$ such that $\mathcal{B}; \emptyset \vdash \mathtt{b} : \overline{T} \to T$ and any $\mathcal{B}; \emptyset \vdash \overline{\mathtt{v}} : \overline{T}$, $\Theta = \overline{\delta} \mapsto \overline{\mathtt{v}}'\langle\rangle$ with $\mathcal{B}; \emptyset \vdash \overline{\mathtt{v}}' : T$, $\delta' \in \{\delta, \overline{\delta}\}$, then we require $(\!|\mathtt{b}|\!)^{\delta,\delta'}_{\Theta,\sigma}$ to be a value of type T.

Theorem 1 (Type Preservation)**.** *Assume that the interpretation of built-in operators respects the given types. Let* $\mathcal{A} = \overline{\mathtt{x}} : \overline{T}$ *and* $\mathcal{D}; \emptyset \vdash \overline{\mathtt{v}} : \overline{T}$*, so that* $\mathit{length}(\overline{\mathtt{v}}) = \mathit{length}(\overline{\mathtt{x}})$*. If* $\mathcal{D}; \mathcal{A} \vdash \mathtt{e} : T$*,* $\Theta \in \mathit{WFVTE}(\mathcal{D}; \mathcal{A}; \mathtt{e})$ *and* $\delta, \delta'; \Theta; \sigma \vdash \mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] \Downarrow \theta$*, then* $\theta \in \mathcal{C}^T_{\mathcal{D};\mathcal{A}}[\mathtt{e}]$*.*

Proof. See [Cas+20]. □

Notice that, since the evaluation of $\mathtt{e}$ produces a value-tree which is coherent with $\mathtt{e}$, the value-tree environment Θ can be proved to be coherent with the main expression by induction on the network evolution.

7.4.2 HFC, HFC′ and Aligned FScaFi

The syntax of HFC is given in Figure 7.9. Its operational semantics is given as a transition system analogous to that in Section 7.3.4, but based on a different judgement for the device operational semantics $\delta; \Theta; \sigma \vdash \mathtt{e}_{\mathtt{main}} \Downarrow \theta$, for which we refer to [Aud+19]. The two main differences are as follows:

- in HFC, values are divided into *local values* and *neighbouring values* (the latter are not allowed to appear in source code), while in FScaFi there are no neighbouring values;

P	::=	$\overline{\texttt{F}}$ e	program
F	::=	def d($\overline{\texttt{x}}$) {e}	function declaration
e	::=	x $\mid$ v $\mid$ ($\overline{\texttt{x}}$) $\overset{\tau}{\texttt{=>}}$ e $\mid$ e($\overline{\texttt{e}}$) $\mid$ rep(e){(x) => e} $\mid$ nbr{e}	expression
v	::=	$\ell \mid \phi$	value
ϕ	::=	$\overline{\delta} \mapsto \overline{\ell}$	neighbouring field value
ℓ	::=	c($\overline{\ell}$) $\mid$ f	local value
f	::=	b $\mid$ d $\mid$ ($\overline{\texttt{x}}$) $\overset{\tau}{\texttt{=>}}$ e	function value
T	::=	$t \mid R \mid L$	type
L	::=	$l \mid S \mid (\overline{T}) \to R$	local type
R	::=	$r \mid S \mid F$	return type
S	::=	$s \mid B \mid (\overline{T}) \to S$	local return type
F	::=	field(S)	neighbouring type

Figure 7.9: Syntax of programs, values and types of HFC.

- in HFC a language construct for `foldhood` is not needed, since a built-in with the same meaning can be defined.

In both of the languages, branching statements are considered as syntactic sugar; however, the keyword commonly used in FScaFi is `branch` while in HFC is `if`. We shall use the same built-in functions for both languages.

Neighbouring values (i.e., maps $\overline{\delta} \mapsto \overline{\ell}$ from device identifiers to local values) are produced in HFC by the `nbr` construct and some built-in functions. HFC distinguishes between types for *local* values from those that are not (namely, neighbouring types F for neighbouring values), as well as between types that are allowed to be *returned* by functions from those that are not. In summary, this induces four different type categories: types T, local types L, return types R, and local return types S. More specifically, the main restrictions enforced by the HFC type system in [Aud+19] in order to ensure the *domain alignment* property[8] are:

[8]Domain alignment holds iff the domain of neighbouring values ϕ obtained from expressions `e` is equal to the set of all neighbours which computed the same `e` in their previous evaluation round.

HFC		*FSCAFI*
`def d(`$\overline{\texttt{x}}$`) {e}`	$\longleftrightarrow$	`def d(`$\overline{\texttt{x}}$`) = @@{e}`
$(\overline{\texttt{x}})$ $\overset{\tau}{\texttt{=>}}$ `e`	$\longleftrightarrow$	$(\overline{\texttt{x}})$ $\overset{\tau}{\texttt{=>}}$ `@@{e}`
`foldhood(e, e, e)`	$\longleftrightarrow$	`foldhood(e)(e){e}`

Figure 7.10: Informal description of the bidirectional translation between HFC and FScaFi.

- anonymous functions cannot capture variables of neighbouring type;
- `rep` statements are demanded to have local return type;
- neighbouring types can only be built from local return types $F = \texttt{field}(S)$, since neighbouring values need to be aggregated and this is possible only for return types, and avoiding "neighbouring values of neighbouring values" which may lead to unintentionally heavy computations;
- types of the form $(\overline{T}) \to F$ (functions returning neighbouring values) are not return types. Thus, functions of type $(\overline{T}) \to F$ are used almost as in a first-order language. In particular, there is no way to write a non-constant expression `e` evaluating to such a function.

The syntaxes of HFC and FScaFi are very similar: the simple rules in Figure 7.10 can translate programs from one syntax to the other, assuming that `foldhood` is the name of a valid HFC built-in and all other built-in names are in common. However, these rules do not generally preserve type-safety and behaviour for *all* programs: they do it for a fragment of the two languages, which we call *HFC′* and *Aligned* FScaFi.

HFC′ is obtained by adding the following three custom restrictions, on how field values can be processed, to the rules of the refined Hindley-Milner types for HFC [Aud+19] described above:

R1 Expressions of neighbouring type can only be aggregated to local values with a `foldhood` operator if they do not capture variables of neighbouring types; so that, e.g., aggregating arguments of neighbouring type is never allowed.

R2 Functions `f` with arguments of neighbouring type have to return a neighbouring type.

R3 Built-in functions need to be pointwise or aggregating on neighbouring values

(as will be formally specified in Definition 3).

Notice that functions with neighbouring arguments and local return type could not compute their return value from their arguments by Restriction R1, and thus would be forced to ignore them. It follows that restriction R2 does not eliminate any further meaningful programs. We also remark that all HFC programs considered in previous works [Aud+19; Aud+18; Vir+18] actually belong to HFC′ (or can easily be reformulated in order to do so).

Example 7.9. *In order to show the rationale behind Restriction R1, consider the following HFC program*

```
def wrong_avghood(x) = {
  foldhood(0, +, x) / foldhood(0, +, 1)
}
wrong_avghood(nbr{sns-temp()})
```

and its translation in FScaFi

```
def wrong_avghood(x) = @@{
  foldhood(0)(+){x} / foldhood(0)(+){1}
}
wrong_avghood(nbr{sns-temp()})
```

We may suppose the FScaFi *program to calculate the average temperature of neighbours, as the HFC program does. Instead, this program is fully equivalent to the simpler program* `nbr{sns-temp()}`*. If we evaluate the main expression against a neighbour δ', we obtain as argument the temperature t of that neighbour. When function* `wrong_avghood` *is applied to t, the neighbour device δ' is ignored by both* `foldhood` *statements, which fail to interpret the captured neighbouring value as such. The value of the function is then $nt/n = t$, where n is the number of neighbours.*

We remark that an FScaFi program computing the average temperature of neighbours could still be conveniently written, by resorting to the programming patterns that will be discussed in Section 7.4.3 and that are idiomatic in ScaFi (e.g., for the example above, it is sufficient to make `x` a by-name parameter).

Example 7.10. *In order to show the rationale behind Restriction R2, consider the following HFC program (where* `if`$(\mathtt{e}_1)\{\mathtt{e}_2\}\{\mathtt{e}_3\}$ *is short for*

$\texttt{mux}(\texttt{e}_1, \texttt{() =>e}_2, \texttt{() =>e}_3)())$

```
def wrong_ignore(x) = { 1 }
foldhood(0, +, if (sns-temp() > 0) { wrong_ignore(nbr{sns-temp()}) } { 1
    } )
```

and its translation in FScaFi

```
def wrong_ignore(x) = @@{ 1 }
foldhood(0)(+){ branch (sns-temp() > 0) { wrong_ignore(nbr{sns-temp()})
   } { 1 } }
```

We may suppose the FScaFi *program to always calculate the total number of neighbours (*$\texttt{foldhood}(0)(\texttt{+})\{1\}$*) as the HFC program does. However, if the sensed temperature is positive, this program only counts neighbours with positive temperature, since the argument* $\texttt{nbr}\{\texttt{sns-temp}()\}$ *fails its evaluation against neighbours with negative temperature.*

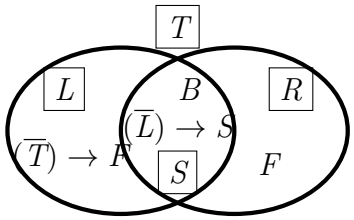
Example 7.11. *In order to show the rationale behind Restriction R3, consider a built-in function* `sorthood` *rearranging values $\phi(\delta)$ relative to neighbours in increasing order of neighbour identifier δ. Formally, applying this function to a neighbouring value $\phi = \overline{\delta} \mapsto \overline{\ell}$ (assuming $\delta_1 \leq \ldots \leq \delta_n$), we obtain the neighbouring value $\phi' = \delta_1 \mapsto \ell_{\pi_1}, \ldots, \delta_n \mapsto \ell_{\pi_n}$ where the permutation π is such that $\ell_{\pi_1} \leq \ldots \ell_{\pi_n}$. This function is conceivable (although artificial) in HFC, but it is not implementable in* FScaFi.

Aligned FScaFi is the fragment of FScaFi that can be typed by rules in Figure 7.11, which enforce the given restrictions. In particular:

- All rules are obtained by translating the corresponding rules for HFC in [Aud+19], and differ from those in Figure 7.3 by acknowledging the existence of the four type categories, by introducing type $\texttt{field}(S)$ in `nbr` statements, and by requiring captured variables to have local type.
- Restriction R1 is further implemented in Rule [T'-FOLD], by requiring each free variable occurring in the third branch to be of local type.
- Restriction R2 is implemented by a change in the syntax of local types (restricting $(\overline{T}) \to R$ to $(\overline{T}) \to F$), as reflected in the Voronoi diagram of types given in Figure 7.11 (top).

Types:

$$
\begin{array}{lcll}
T & ::= & t \mid R \mid L & \text{type} \\
L & ::= & l \mid S \mid (\overline{T}) \to F & \text{local type} \\
R & ::= & r \mid S \mid F & \text{return type} \\
S & ::= & s \mid B \mid (\overline{L}) \to S & \text{local return type} \\
F & ::= & \texttt{field}(S) & \text{field type}
\end{array}
$$



Local type schemes:

$$LS ::= \forall \overline{t}\overline{l}\overline{r}\overline{s}.L \qquad \text{local type scheme}$$

Expression typing: $\boxed{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : T}$

$$\frac{}{\mathcal{D}; \mathcal{A}, \texttt{x} : T \vdash \texttt{x} : T}\ [\text{T'-VAR}] \qquad \frac{S'[\overline{s} := \overline{S}''] = (\overline{S}) \to S \qquad \mathcal{D}; \mathcal{A} \vdash \overline{\ell} : \overline{S}}{\mathcal{D}, \texttt{c} : \forall \overline{s}.S'; \mathcal{A} \vdash \texttt{c}(\overline{\ell}) : S}\ [\text{T'-DAT}]$$

$$\frac{\overline{\texttt{y}} = \mathbf{FV}((\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{@@\{e\}}) \quad \mathcal{D}; \mathcal{A} \vdash \overline{\texttt{y}} : \overline{L} \quad \mathcal{D}; \mathcal{A}, \overline{\texttt{x}} : \overline{T} \vdash \texttt{e} : R}{\mathcal{D}; \mathcal{A} \vdash (\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{@@\{e\}} : (\overline{T}) \to R}\ [\text{T'-A-FUN}]$$

$$\frac{\texttt{f} \text{ is a (built-in or declared) function}}{\mathcal{D}, \texttt{f} : \forall \overline{t}\overline{l}\overline{r}\overline{s}.L; \mathcal{A} \vdash \texttt{f} : L[\overline{t} := \overline{T},\ \overline{l} := \overline{L},\ \overline{r} := \overline{R},\ \overline{s} := \overline{S}]}\ [\text{T'-N-FUN}]$$

$$\frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : (\overline{T}) \to R \qquad \mathcal{D}; \mathcal{A} \vdash \overline{\texttt{e}} : \overline{T}}{\mathcal{D}; \mathcal{A} \vdash \texttt{e}(\overline{\texttt{e}}) : R}\ [\text{T'-APP}]$$

$$\frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e}_1 : S \quad \mathcal{D}; \mathcal{A} \vdash \texttt{e}_2 : S \to S}{\mathcal{D}; \mathcal{A} \vdash \texttt{rep}(\texttt{e}_1)\{\texttt{e}_2\} : S}\ [\text{T'-REP}] \qquad \frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : S}{\mathcal{D}; \mathcal{A} \vdash \texttt{nbr}\{\texttt{e}\} : \texttt{field}(S)}\ [\text{T'-NBR}]$$

$$\frac{\begin{array}{ll} \mathcal{D}; \mathcal{A} \vdash \texttt{e}_1 : S & \mathcal{D}; \mathcal{A} \vdash \texttt{e}_3 : \texttt{field}(S) \text{ or } S \\ \mathcal{D}; \mathcal{A} \vdash \texttt{e}_2 : (S, S) \to S & \mathcal{D}; \mathcal{A} \vdash \overline{\texttt{x}} : \overline{L} \quad \text{where } \overline{\texttt{x}} = \mathbf{FV}(\texttt{e}_3) \end{array}}{\mathcal{D}; \mathcal{A} \vdash \texttt{foldhood}(\texttt{e}_1)(\texttt{e}_2)\{\texttt{e}_3\} : S}\ [\text{T'-FOLD}]$$

Function typing: $\boxed{\mathcal{D} \vdash \texttt{F} : LS}$

$$\frac{\mathcal{D}, \texttt{d} : (\overline{T}) \to R; \overline{\texttt{x}} : \overline{T} \vdash \texttt{e} : R \qquad \overline{t}\overline{l}\overline{r}\overline{s} = \mathbf{FTV}((\overline{T}) \to R)}{\mathcal{D} \vdash \texttt{def d}(\overline{\texttt{x}})\texttt{ = @@\{e\}} : \forall \overline{t}\overline{l}\overline{r}\overline{s}.(\overline{T}) \to R}\ [\text{T'-FUNCTION}]$$

Program typing: $\boxed{\mathcal{D}_0 \vdash \texttt{P} : T}$

$$\frac{\begin{array}{l} \texttt{F}_i = (\texttt{def d}_i(\_)\ \_) \quad \mathcal{D}_{i-1} \vdash \texttt{F}_i : LS_i \quad \mathcal{D}_i = \mathcal{D}_{i-1}, \texttt{d}_i : LS_i \quad (i \in 1..n) \\ \mathcal{D}_n; \emptyset \vdash \texttt{e} : T \end{array}}{\mathcal{D}_0 \vdash \texttt{F}_1 \cdots \texttt{F}_n\ \texttt{e} : T}\ [\text{T'-PROGRAM}]$$

Figure 7.11: Hindley-Milner typing for Aligned FScaFi expressions, function declarations, and programs.

- Restriction R3 is implicitly valid for all FScaFi programs, as it is impossible to define a non-positional built-in operator following the semantic rules.

The embedding of Aligned FScaFi as a fragment of FScaFi can be formally characterised by means of the following definition and theorem.

Definition 2 (Erasure)**.** *The* erasure *of a Aligned* FScaFi *type T is the type* ***erasure***(T) *obtained from T by replacing all occurrences of* `field`(L) *with L and dropping the distinction between the different kinds of type variables (i.e., considering each of them as a standard type variable t). Similarly, the erasure of a type scheme $\forall \overline{t} \overline{l} \overline{r} \overline{s}.L$ is the type scheme $\forall \overline{t} \overline{l} \overline{r} \overline{s}.$**erasure**$(L)$ (dropping distinction between kinds of variables). Finally, the erasure of a type environment $\mathcal{A} = \overline{\mathtt{x}} : \overline{T}$ is* ***erasure***$(\mathcal{A}) = \overline{\mathtt{x}} :$ ***erasure***$(\overline{T})$*; and the erasure of a type-scheme environment is* ***erasure***$(\mathcal{D}) = \overline{\mathtt{x}} :$ ***erasure***$(\overline{LS})$*.*

Theorem 2 (Typing Correspondence)**.** *Assume that $\mathcal{D}; \mathcal{A} \vdash \mathtt{e} : T$ in Aligned* FScaFi*. Then* ***erasure***$(\mathcal{D})$; ***erasure***$(\mathcal{A}) \vdash \mathtt{e} :$ ***erasure***(T) *in* FScaFi*.*

Proof. See [Cas+20]. □

It is worth observing that a corresponding type system for HFC, enforcing restrictions R1 and R2, can be given by translating the syntax through the rules in Figure 7.10. Moreover, for HFC, Restriction R3 is not incorporated in the type system, but can be expressed as an additional coherence assumption between HFC and FScaFi, demanding built-in operators to be *positional* as per the following definition.

Definition 3 (Positional Built-in Operators)**.** *We say that an HFC built-in operator is* positional *if it can be obtained by composition from the following operators:*

- *built-in operators and sensors with local inputs (as* `temperature`, `nbrRange`*), including operator* `consthood(v)`*, which returns a field constantly equal to its (local) input;*
- `map(f,`$\overline{\mathtt{e}}$`)`*, which applies a function* `f` *with local inputs and output (of any ariety) pointwise to fields* $\overline{\mathtt{e}}$*;*
- `foldhood(e,e,e)`*, which collapses a field via an aggregator (exactly as in* FScaFi*).*

Positional operators are characterised by being essentially induced by operators with local inputs. Using the HFC assumption that all local built-in operators are implicitly overloaded to accept any combination of scalar and field parameters by operating pointwise [Aud+19], it follows that a program using positional operators can be rewritten to use only local operators, as per the following lemma.

Lemma 2 (A Normal Form for HFC Programs)**.** *Any HFC program* `P` *using only positional built-in operators can be put in* normal form*, by repeatedly applying the following transformations:*

- *expanding all built-in operators into the basic positional operators listed in Definition 3;*
- *substituting every occurrence of* `consthood(e)` *with* `e`*;*
- *substituting* $\texttt{map}(\texttt{f}, \overline{\texttt{e}})$ *with* $\texttt{f}(\overline{\texttt{e}})$*.*

Proof. The whole transformation is correct since local functions in HFC are implicitly overloaded to accept any combination of scalar and field parameters by operating pointwise [Aud+19]. As a result of the transformation, only built-in operators with local arguments will be present. □

The normal form of an HFC′ program is of particular interest since it enables to translate back this program into Aligned FScaFi, by removing operators that would not be expressible in Aligned FScaFi. In order for an Aligned FScaFi program and its HFC′ translation to have the same behaviour, we need the built-in functions to have that same behaviour, as detailed in the following definition.

Definition 4 (Built-in Coherence)**.** *We say that closed expressions (or programs)* $\texttt{e}_H : T$ *in HFC′ and* $\texttt{e}_S : T$ *in Aligned* FScaFi *have the* same behaviour *whenever:*

- *if* T *is a local type,* $\delta; \Theta; \sigma \vdash \texttt{e}_H \Downarrow \ell\langle\overline{\theta}\rangle$ *if and only if* $\delta, \delta; \Theta; \sigma \vdash \texttt{e}_S \Downarrow \ell\langle\overline{\theta}'\rangle$ *for some* $\overline{\theta}$ *and* $\overline{\theta}'$*;*
- *if* T *is a field type,* $\delta; \Theta; \sigma \vdash \texttt{e}_H \Downarrow \phi\langle\overline{\theta}\rangle$ *if and only if* $\delta, \delta_i; \Theta; \sigma \vdash \texttt{e}_S \Downarrow \ell_i\langle\overline{\theta}'\rangle$ *with* $\phi = \delta_i \mapsto \ell_i$ *for some* $\overline{\theta}$ *and* $\overline{\theta}'$*.*

We say that HFC′ and Aligned FScaFi *are* built-in coherent *iff for every built-in operator* `b` *with local arguments,* $\texttt{b}(\overline{\ell})$ *has the same behaviour in both languages.*

Assuming that built-in operators are positional, the coherence assumption above and well-typedness according to the type system in Figure 7.11, we are now able to prove that the proposed translation preserves the program behaviour.

Theorem 3 (Equivalence between HFC′ and Aligned FScaFi)**.** *Assume that HFC′ and Aligned* FScaFi *are built-in coherent. If* $\mathtt{P}^H$ *is a well-typed HFC′ program in normal form without functions with field arguments*[9]*, then its translation* $\mathtt{P}^S$ *is a valid Aligned* FScaFi *program with the same behaviour (i.e., such that* $\delta;\Theta;\sigma \vdash \mathtt{e}^H_{\mathtt{main}} \Downarrow \theta$ *if and only if* $\delta,\delta;\Theta;\sigma \vdash \mathtt{e}^S_{\mathtt{main}} \Downarrow \theta$*). Conversely, if* $\mathtt{P}^S$ *is a valid Aligned* FScaFi *program, then its translation* $\mathtt{P}^H$ *is a well-typed HFC′ program with the same behaviour (i.e., such that* $\delta;\Theta;\sigma \vdash \mathtt{e}^H_{\mathtt{main}} \Downarrow \theta$ *if and only if* $\delta,\delta;\Theta;\sigma \vdash \mathtt{e}^S_{\mathtt{main}} \Downarrow \theta$*).*

Proof. See [Cas+20]. □

7.4.3 FScaFi expressiveness

In this section, we argue that FScaFi is an expressive language for distributed computations. Section 7.4.3 shows that Aligned FScaFi contains most relevant programs and is in fact universal for distributed computations. Section 7.4.3 presents few programming patterns, enabling to conveniently express most programs so that they belong to Aligned FScaFi. Section 7.4.3 argues that the FScaFi programs that are not in Aligned FScaFi can fruitfully extend the expressive power of HFC.

Universality and self-stabilisation in Aligned FScaFi The correspondence between Aligned FScaFi and HFC′ given by Theorem 3 enables, as byproduct, the direct transfer to FScaFi of important HFC properties, like the following two properties.

Turing Universality [Aud+18]. A programming model for distributed systems is Turing-universal if and only if it is able to replicate the behaviour of any Turing machine, which in every event takes as input the whole collection of causally available data.

[9] Thus using only built-in operators with local arguments.

```
s ::= x | v | f(s̄) | branch(s){s}{s} | nbr{s}
    | rep(e){(x)=> @@{f^C(x, s, ē)}}
    | rep(e){(x)=> @@{f(foldhood(s)(f){mux(nbrlt(s), nbr{x}, s)}, s̄)}}
    | rep(e){(x)=> @@{f^R(foldhoodPlus(s)(min){f^MP(nbr{x}, s̄)}, x, ē)}}
```

Figure 7.12: Syntax of a self-stabilising fragment of field calculus expressions, where self-stabilising expressions `s` occurring inside a `rep` statement cannot contain free occurrences of the `rep`-bound variable `x`.

Self-Stabilisation [Vir+18]. A time- and space-distributed data is *stabilising* iff it remains constant in every point after a certain time t_0, and its *limit* is the value assumed after t_0. A distributed program is *self-stabilising* iff given stabilising inputs and topology, it produces a stabilising output which depends only on the limits of the inputs and topology (and not on the concrete scheduling of events, nor on the input values before stabilisation).

In fact, Aligned FScaFi is Turing universal (assuming a sufficient collection of built-ins), since HFC can be proved to be so [Aud+18] through a program whose translation belongs to the fragment identified by the restricted type system (Figure 7.11).

Similarly, the programming patterns provided in [Vir+18] for designing self-stabilising applications can be rewritten in order to fit into the fragment, obtaining the fragment in Figure 7.12 identifying self-stabilising Aligned FScaFi programs.

Programming patterns The restrictions imposed by the type system in Figure 7.11 are subtle, and programming according to them may require some forethought. In particular, there may be issues in programming functions with field arguments, and folding field parameters. However, there are few programming patterns that we can use in order to comply with it.

1. *Abstracting*: a field-like argument may be passed "by name" through

$$((\texttt{x})\texttt{=> @@}\{\texttt{e}_1\})(\texttt{e}_2) \quad \longrightarrow \quad ((\texttt{x})\texttt{=> @@}\{\texttt{e}_1[\texttt{x} := \texttt{x}()]\})(()\texttt{=> @@}\{\texttt{e}_2\})$$

2. *Deferring*: an `nbr` in the argument may be transferred into the body as

$$((\texttt{x}) \texttt{ => } \texttt{@@}\{\texttt{e}_1\})(\texttt{nbr}\{\texttt{e}_2\}) \quad \longrightarrow \quad ((\texttt{x}) \texttt{ => } \texttt{@@}\{\texttt{e}_1[\texttt{x} := \texttt{nbr}\{\texttt{x}\}]\})(\texttt{e}_2)$$

Both rewrites convert a field argument into a local argument, thus allowing its capture into `foldhood` statements, possibly avoiding the additional restrictions on functions with field arguments. Using these rewrites, we are able to cover the following types of function applications $\texttt{f}(\overline{\texttt{e}})$:[10]

- If the field parameter is never folded, no rewrite is needed, and the function correctly performs a point-wise operation both in HFC and in FScaFi.
- If the field parameter is an expression $\texttt{e}_2$ whose computation does not depend on the current domain (e.g., relational sensors as `nbrRange`), we can perform rewrite (1) without modifying the behaviour.
- Rewrite (1) is also correct provided that no occurrence of `x` is inside a branching statement: in this case, it is granted that `x`() will be computed against the same environment as it would have been expression $\texttt{e}_2$, thus producing the same result.
- If the field parameter is directly obtained from an `nbr` statement, we can perform rewrite (2) which ensures that the fields used in the function body correctly correspond to the results of expression $\texttt{e}_2$ *as computed in the larger domain* of all devices evaluating the function call.

The remaining problematic case is whenever the argument $\texttt{e}_2$ is a field expression not writable as an `nbr` which depends on the current domain, and which is passed into a function $\texttt{e}_1$ folding it *inside a branching statement*. We believe that this situation is rare, often avoidable, and in fact it does not occur in any of the examples of program classes ever proposed for HFC, which we will partially discuss in the following section.

FScaFi programs that go beyond HFC programs As argued in [Aud+16], programs such as *updatable metrics* and *combined Boolean restriction* are not conveniently expressed in HFC. In the former case, we can use the following general

[10] Here we assume that the function to be applied is a value. If it is not, further restrictions have to be taken into account (we do not present them to keep the presentation simpler).

scheme for updatable functions, first proposed in [Aud+19]:

```
def up(injecter) = @@{
  snd( rep(injecter()) {
    (x) => @@{ foldhood(injecter())(max){nbr{x}} }
} ) }
```

where `injecter` is a function returning a pair ⟨`version number`, `function code`⟩, and the built-in operator `max` selects the pair with the highest version number among its arguments. This procedure defines a perfectly reasonable "upgradeable function" by spreading functions with higher version number throughout devices. However, it is not allowed by the type system of HFC for functions returning fields, such as metrics (which usually have type () → `num`). This scheme can instead be used in FScaFi (as shown in Section 7.6), and works properly provided that new versions are injected at a slow rate, and an occasionally empty domain of a field-like expression does not produce critical effects.

Another situation where the permissive behaviour of FScaFi is crucial is that of *combined Boolean restriction*. In this setting, a field-like value `x` need to be restricted to those devices agreeing on the value of n Boolean parameters $\mathtt{b}_1, \ldots, \mathtt{b}_n$, before being processed by a function `f`. This rather abstract example might be concretely instantiated, e.g., in case a function needs to be executed separately on devices with different configurations. In HFC, this effect can be achieved only by restricting on each of the 2^n possibilities for the parameters, as in the following.

```
if (b1 && b2 && ... bn) {f(x)} {
  if (!b1 && b2 && ... bn) {f(x)} {
    if (b1 && !b2 && ... bn) {f(x)} { ... }}}
```

However, such a program might be infeasibly large even for small values of n. On the other hand, in FScaFi the above program can be concisely rewritten as:

```
f(x + branch (b1) {nbr{0}} {nbr{0}} + branch (b2) {nbr{0}} {nbr{0}} +
    ...)
```

whose size is linear in n.[11] The domain of the i-th 0-valued field-like subexpression

[11] In this code we assumed that `x` has numerical type, but similar code can be obtained for any type by defining a binary operator which is the identity on its first argument.

above is equal to the set of devices agreeing on $\mathtt{b}_i$, hence by intersecting all of them the resulting domain corresponds to the set of devices agreeing on each of the n given parameters.

7.5 ScaFi: Library

In this section, we show how ScaFi can be used to implement typed aggregate APIs (such as the key operators introduced in Section 5.2.2). Notably, the type class idiom can be used to formalise certain requirements on inputs and accordingly constrain at compile-time the use of functions to suitable parameters.

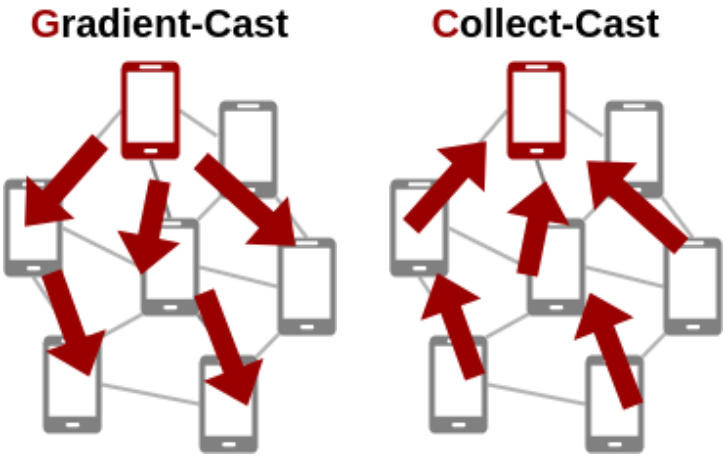
7.5.1 Fundamental building blocks

Consider Figure 7.13.

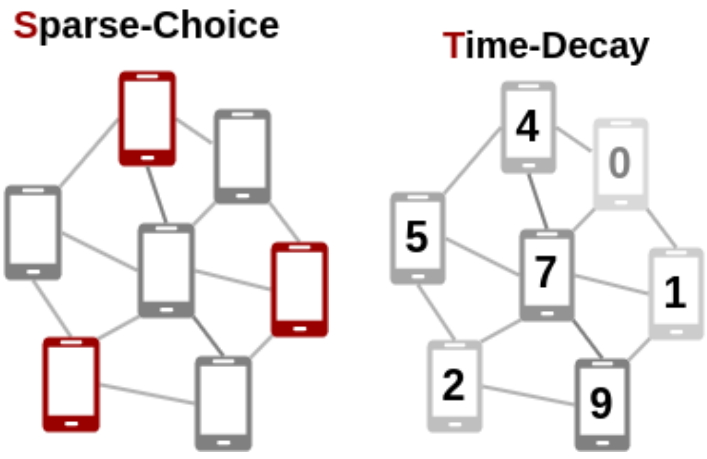
Gradient-cast Multiple types of aggregations can be performed along a distance-gradient. In fact, it comes handy to define a generalised operator `G` (Figure 7.13a) as a gradient algorithm parameterised upon the `metric` for calculating increments (i.e., distances), which can carry some value of `field` from the source outward, with the logic `acc` by which such value gets evolved while ascending the gradient:

```
def G[V:OrderingFoldable](src:Boolean, phi:V, acc:V=>V, metric: =>Double): V =
  rep((Double.MaxValue,phi)){ case (distance,value) =>
    mux(src) {
      (0.0, phi)   // ..on sources
    } {
      minHoodMinus { // minHood except myself
         (nbr{ distance } + metric, acc( nbr{ value }))
      }
    }
  }._2 // yielding the resulting field of values
```

The context bound `V:OrderingFoldable` statically enforces that the instantiated generic type `V` has an implicit `OrderingFoldable[V]` typeclass instance in scope, which provides a definition of methods `top():V`, `bottom():V`, and `compare(V,V):Int`. These constraints on `V` ensure that `minHoodMinus` can work out the minimum value for tuples `(distance,value)`, where we also assume that



(a) Information propagation. (b) Information collection.



(c) Leader election. (d) State and time.

Figure 7.13: Basic aggregate building blocks.

in the scope of the definition of `G` there are implicit `OrderingFoldable`s for both `Double`s and 2-element tuples of `OrderingFoldable`s.

Upon `G`, it is straightforward, for example, to implement a basic `hopGradient` (where a distance is the number of hops from a node to another) and a `broadcast` function that simply propagates a value from source points to the rest of the network:

```
def hopGradientByG(src: Boolean): Int =
  G[Int](src, 0, acc = _+1, metric = 1)

def broadcast[V:OrderingFoldable](source: Boolean, field: V): V =
  G[V](source, field, acc = x=>x, metric = nbrRange)
```

Converge-cast Essentially, `G` allows for an information flow from source devices to their global surroundings—a sort of propagation or diffusion of values. The dual operation involves an information flow directed from a global area towards specific collection points, which can be used to perform distributed sensing. This is supported by the generalised operator `C` (Figure 7.13b), which `accumulates` values along the `potential` field, starting with `local` at the sources where `potential` is maximum and aggregating while descending the chain of parents, ultimately converging to the points where `potential` is minimum.

```
def C[V:OrderingFoldable](potential: V, acc: (V,V)=>V, phi: V, Null: V): V = {
  rep(phi){ v =>
    acc(phi, foldhood(Null)(acc){
      mux(nbr(findParent(potential)) == mid()){ nbr(v) } { nbr(Null) }
    })
  }
}

def findParent[V:OrderingFoldable](p: V): ID = {
  mux(implicitly[OrderingFoldable[V]].compare(minHood{ nbr(p) }, p)<0 ){
    minHood{ nbr{ (p, mid()) } }._2
  }{ Int.MaxValue }
}
```

To better visualise how the algorithm works, let's consider a 3×3 grid of devices with unitary distance between rows and columns, neighbouring relation on adjacent rows and columns (i.e., Manhattan distance), and device 3 at the 2nd row and 1st column with the `"source"` sensor set to `true`. The following expression:

```
def p = distanceTo(isSource) // potential

(p, mid()+"->"+findParent(p), C[Double](p, _+_, 1, 0.0))
```

evaluates to

```
/* (1, 0->3,  3)    (2, 1->0, 2)    (3, 2->1, 1)
   (0, 3->.., 9)    (1, 4->3, 4)    (2, 5->4, 2)
   (1, 6->3,  1)    (2, 7->4, 1)    (3, 8->5, 1) */
```

as, for example, the source device (where the potential field is 0) folds (with a sum) on the aggregated values coming from the top, bottom, and right devices; conversely, "edge" devices (with no "parent") at distance `3` from the source emit the `local` value `1`.

Sparse-choice The generic operator `S` (Figure 7.13c) enables to select devices sparsely in such a way that the network gets partitioned into "areas of responsibility". In other words, it carries out a leader election process (see Figure 7.14), where `grain` is the mean distance between two leaders—according to a notion of distance expressed by `metric`. It could be implemented as follows:

```
def S(grain: Double, metric: Double): Boolean =
  breakUsingUids(randomUid, grain, metric)
```

where `randomUid` generates a random field of unique identifiers:

```
def randomUid: (Double,ID) =
  rep((Math.random()), mid()) { v => (v._1, mid()) }
```

which is in turn exploited to break the network symmetry:

```
def breakUsingUids(uid: (Double,ID), grain: Double, metric: => Double): Boolean
    =
  uid == rep(uid) { lead:(Double,ID) =>
    val acc = (_:Double)+metric
    distanceCompetition(G[Double](uid==lead,0,acc,metric),
                        lead, uid, grain, metric)
  }
```

by means of a competition for leadership between devices defined as:

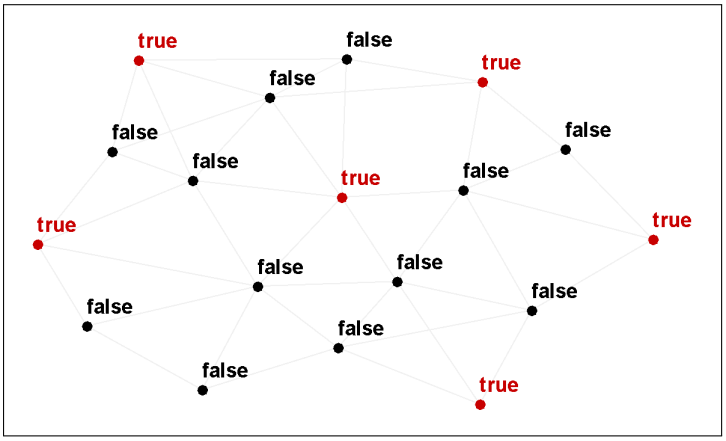


Figure 7.14: Stabilised field in a ScaFi simulation for `S`. The red nodes are those which compute `true`, i.e., the elected leaders.

```
def distanceCompetition(d: Double, lead: (Double,ID), uid: (Double,ID),
                        grain: Double, metric: => Double) = {
  val inf:(Double,ID) = (Double.PositiveInfinity, uid._2)
  mux(d > grain){ uid }{
    mux(d >= (0.5*grain)){ inf }{
      minHood {
        mux(nbr{d}+metric >= 0.5*grain){ nbr{inf} }{ nbr{lead} }
} } } }
```

Time-decay The `T` operator (Figure 7.13d) can be used to express time-related patterns, providing a convenient abstraction over `rep` construct. It works by decreasing the `initial` field with a `decay` function until a `floor` value is reached:

```
def T[V:Numeric](initial: V, floor: V, decay: V=>V): V = {
  val ev = implicitly[Numeric[V]] // getting a Numeric[V] object from the
    context
  rep(initial){ v =>
    ev.min(initial, ev.max(floor, decay(v)))
  }
}
```

Upon `T`, the implementation of a `timer` function is straightforward:

```
def timer[V](initial: V): V = {
  val ev = implicitly[Numeric[V]] // getting a Numeric[V] object from the
    context
  T(initial, ev.zero, (t:V)=>ev.minus(t, ev.one))
} // Decreases 'initial' by 1 at each round, until 0
```

In turn, `timer` supports the definition of a `limitedMemory` function that computes `value` for `timeout` and then returns `expValue` after expiration, effectively realising a finite-time memory.

```
def limitedMemory[V,T](value: V, expValue: V, timeout: T): (V,T) = {
  val ev = implicitly[Numeric[V]] // getting a Numeric[V] object from the
    context
  val t = timer[T](timeout)
  (mux(ev.gt(t, ev.zero)){value}{expValue}, t)
}
```

Note that the above definition of `timer` depends on the frequency of operation of a given device. If one desires a notion of temporariness that is based on physical time, it could be implemented as follows:

```
def timer(dur: Duration): Long = {
  val ct = System.nanoTime()  // Current time
  val et = ct + dur.toNanos   // Time-to-expire (bootstrap)

  rep((et, dur.toNanos)) { case (expTime,remaining) =>
    mux(remaining<=0) { (et,0) }{ (expTime, expTime - ct) }
  }._2 // Selects the component expressing remaining time
}
```

where the state about both the expiration time and the remaining time is retained across rounds via `rep`. A simulation for `timer` is shown in Figure 7.15.

7.5.2 Proof of concept: library support for explicit fields

In order to show the expressiveness of ScaFi and the benefit of the Scala integration, we provide a brief practical account of the translation of HFC into FScaFi illustrated in Section 7.4.3. Indeed, thanks to the flexibility of Scala and the minimality of the FScaFi model, ScaFi is able to seamlessly support explicit fields through a small object-functional library[12]. In particular, we can represent

[12]Here, we merely focus on functionality and the key semantic aspects, largely neglecting performance considerations.

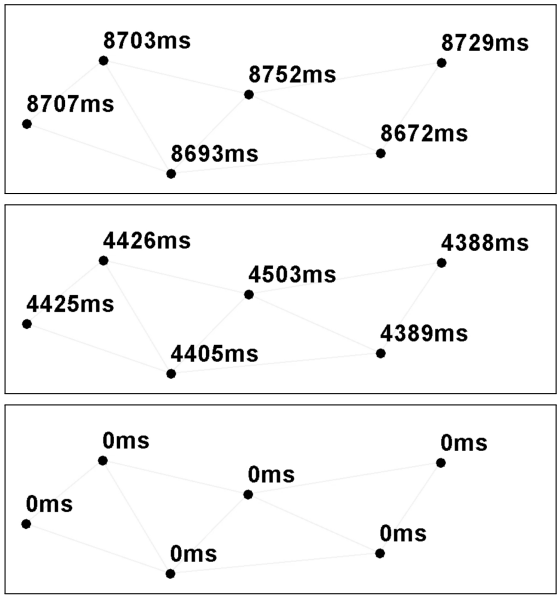


Figure 7.15: Snapshots of a ScaFi simulation for `timer`; the third one depicts the stabilised field.

fields according to their definition as a class wrapping a map from device identifiers to `T` values.

```
class Field[T](val m: Map[ID,T]) { ... }
object Field { // companion object
  def apply[T](m: Map[ID,T]) = new Field(m) // factory method
}
```

Then we just need a way to lift local values and neighbouring sensor queries into field values:

```
def fnbr[A](e: => A): Field[A] =
  Field[A](reifyField(nbr(e)))

def fsns[A](e: => A): Field[A] =
  Field[A](reifyField(e))
```

where we leverage the following function to reify an implicit ScaFi field:

```
def reifyField[T](expr: => T): Map[ID, T] =
  foldhood(Map[ID, T]())(_ ++ _) {
    Map(nbr { mid() } -> expr)
  }
```

Then, for typical operations on fields (such as mapping and folding), we can provide

suitable combinators in the `Field` class:

```
def map2i[R,S](f: Field[R])(o: (T,R)=>S): Field[S] =
  Field(this.restricted.m.collect { case (i,v) if f.m.contains(i) => i -> o(v,f
    .m(i)) })

def fold[V>:T](z:V)(o: (V,V)=>V): V =
  this.restricted.m.values.fold(z)(o)
```

where, crucially, the domain of the field `this` must be properly restricted (and possibly intersected with other fields involved) to the current domain in the program, e.g., through a method like the following.

```
def restricted: Field[T] = {
  val alignedField = fnbr{1} // Build a field to find current domain
  Field(m.filter(el => alignedField.m.contains(el._1))) // Filter by looking at
     that domain
}
```

We can also provide syntactic sugar for specific types, e.g., numeric ones:

```
implicit class NumericField[T:Numeric](f: Field[T]){
  private val ev = implicitly[Numeric[T]]

  def +(f2: Field[T]): Field[T] = f.map2i(f2)(ev.plus(_,_))
  // ...
}
```

through static-time implicit instantiation of the corresponding extension class (which is generic and requires a typeclass instance `Numeric[T]` available in scope) upon attempt to invoke a method like `+` which is not available on the basic `Field` type. Implicit conversions may also be used to simplify passing from local values to field values and viceversa:

```
implicit def localToField[T](lv: T): Field[T] =
  fnbr(mid).map(_ => lv)

implicit def fieldToLocal[T](fv: Field[T]): T =
  fv.m(mid)
```

Finally, the following example provides a taste of this library in action.

```
def gradient(source: Field[Boolean]): Field[Double] = // signature with
    explicit fields
  rep(Double.MaxValue){ // we use standard "local" rep
    d => mux(source) { 0.0 } {
      (fnbr(d) + fsns(nbrRange)).minHoodPlus // builds explicit field and then
    folds into local
    }
  } // automatic local-to-field conversion

def main(): Double =
  gradient(sense[Boolean](SRC)) // automatic local-to-field conversion
  // automatic field-to-local conversion for return
```

7.6 Case Study

ScaFi has been applied to various distributed computing applications [CAV18; CV18; Cas+19b; CV19; Cas+19a]. Here, we show a case study in computational trust (Section 7.6.1), where we address a security vulnerability in collective algorithms (though the same principles may be used to tackle any kind of deviance, e.g., related to failure or performance of peers). Then, Chapters 8 and 10 contain more use cases adopting ScaFi, e.g., for situated problem solving and resource orchestration.

7.6.1 Computational trust for attack-resistant gradients

On computational trust In several community-based domains, it is not possible nor convenient to maintain a trustworthiness infrastructure relying on centralised trusted third parties. To overcome such a limitation, trust relations are typically constructed on the base of direct observations and, possibly, recommendations gathered by interacting with the neighbourhood. For instance, in trustworthy crowdsourcing and sensor networks, a computational notion of trust derives from the exchange and aggregation of information disseminated by the participating nodes [GBS08; Yu+12; Han+14; Mou+15; BB04]. Once a trust metric is established, the usual trust-based decision-making policy consists of comparing the trust estimated by a node, called *trustor*, about the expected behaviour of another node, called *trustee*, and a *trust threshold value tth*, which may depend on several

subjective factors, like, e.g., the initial willingness of the trustor to cooperate with the (possibly unknown) trustee.

In the computational trust literature, several metrics are based on a Bayesian approach. In essence, the trustor assumes that there exists an unknown parameter θ used to predict probabilistically the future good/bad behaviour of the trustee, and the related outcome is drawn independently for each interaction between them. In order to model uncertainty, θ is drawn by a given *prior* distribution, updated as new interactions between the parties occur. Among the various probability prior distributions proposed in the literature, the beta distribution received particular attention [JI02; BB04; GBS08; PKK16]. Such a distribution is fed with two parameters, α and β, which count the number of positive and negative observations experienced by the trustor when interacting with the trustee, respectively. The evaluation of each observation depends on the context. As an example, in the setting of data relaying, a packet sent from the trustor that is forwarded (resp., discarded) by the trustee is considered as a positive (resp., negative) cooperation.

Then, trust is estimated as the statistical expectation `E` of a beta distribution Beta parameterised with respect to α and β, by assuming the initial scenario $\alpha = \beta = 0$, denoting absence of any prior interaction between the parties. Formally:

$$\texttt{E}(\text{Beta}(\alpha + 1, \beta + 1)) = \frac{\alpha + 1}{\alpha + \beta + 2}.$$

Notice that the initial trust is equal to 0.5, which expresses a situation of total uncertainty about the expected behaviour of the trustee.

Different techniques based on such a Bayesian approach differ for the way in which *(i)* observations are weighted, e.g., depending on their age, and *(ii)* recommendations gathered via interactions with the neighbours are combined with the parameters discussed above.

The ageing mechanism can be implemented either by decreasing periodically the result of past observations by a weight w, or by assuming that only the last n observations contribute to the computation of trust. This kind of mechanism avoids the past behaviour to be too impairing over the current behaviour, thus mitigating the effect of on-off misbehaviours. Notice that we will consider the latter approach in the application to Aggregate Computing.

On the other hand, the recommended values received by a node from the neigh-

bourhood, and related to the trust towards a specific trustee, are somehow combined to contribute to the computation of the subjective trust of the node towards the trustee. To avoid subtle colluding attacks, like, e.g., bad mouthing (fake negative recommendations about a honest node) and ballot stuffing (fake positive recommendations about a malicious node), the aggregation privileges direct observations with respect to evidences obtained from other nodes and weights such evidences by the trust towards the nodes providing them.

Application in aggregate computing The Bayesian approach surveyed above can be applied also to the fully-distributed computational framework of Aggregate Computing. For the sake of simplicity, we consider the case in which nodes compute locally on the base of numerical values exchanged with the neighbours, as in the case, e.g., of the gradient field. In Aggregate Computing, the trustworthiness of the nodes depends on the quality of the information they share at each round. Hence, the two trust parameters α and β shall reflect such a relation. In order to estimate the quality of shared data, we first observe the following basic principle: if all the nodes are cooperative, the estimates of the gradient that every node receives from its neighbourhood in a round shall not be too much different from each other, up to certain fluctuations that may depend on several factors, like, e.g., the topology of the network, the location of the source node, the firing frequency of each node, and so on. Hence, if the value received from a node differs too much from the others, then such an observation is used to impair negatively the trust towards that node. In other words, unexpected perturbations of the gradient against the overall trend are considered as a potential attack to the system. On the other hand, a gradient estimate matching the general trend of the gradient field represents a good observation that can be used to affect positively the trust towards the node providing that value.

In order to implement this idea, we estimate trust by following the Bayesian approach in such a way that every gradient estimate received in a round is compared with the average of all the estimates received in that round. The detected difference is then used to evaluate the observation and update the parameters feeding the trust metric, by assuming that if the difference is evaluated positively (resp., negatively) then parameter α (resp., β) is increased. Since the mean square

deviation, called s, represents a standard way to predict differences among values, we use it as the basis to compute the tolerance threshold, called *maxError*, for the evaluation of the difference above. In particular, we assume that the tolerance threshold is computed as a function dependent on s, whose definition represents a parameter of the trust system used to determine whether to deliver penalties, by increasing β, or rewards, by increasing α.

Once α and β are updated according to the policy above, they are possibly integrated with recommendations provided by the neighbourhood by using a mechanism inspired by [JI02; GBS08]. Then, the resulting pair of updated parameters feeds the Beta distribution that governs the computation of the trust metric, which is then compared against the trust threshold. Finally, only the gradient estimates received from trusted nodes are actually used to update the local estimate of the gradient.

Formally, each node i maintains locally the pair of parameters $(\alpha_{ij}, \beta_{ij})$ for each neighbour j. Their initial value is zero. At each round, node i performs the following operations:

1. Node i computes the mean $\bar{x}_i$ of the values x_{ij}, $1 \leq j \leq N$, read from the N neighbours that have a value to communicate and then, assumed the deviation $\xi_{ij} = x_{ij} - \bar{x}_i$, computes the mean square deviation:

$$s = \sqrt{\frac{\sum_{j=1}^{N} \xi_{ij}^2}{N}}.$$

2. For each neighbour j, if $|x_{ij} - \bar{x}_i| > \mathit{maxError}$ then $\beta_{ij} = \beta_{ij} + 1$, else $\alpha_{ij} = \alpha_{ij} + 1$.
3. If the recommendation mechanism is enabled, for each neighbour j, node i receives from any other node k in the neighbourhood the pair $(\alpha_{kj}, \beta_{kj})$, and then computes the following recommended values:

$$\alpha_j^{\text{rec}} = \sum_{1 \leq k \leq N, k \neq i,j} \frac{2 \cdot \alpha_{ik} \cdot \alpha_{kj}}{(\beta_{ik} + 2) \cdot (\alpha_{kj} + \beta_{kj} + 2) + 2 \cdot \alpha_{ik}}$$

$$\beta_j^{\text{rec}} = \sum_{1 \leq k \leq N, k \neq i,j} \frac{2 \cdot \alpha_{ik} \cdot \beta_{kj}}{(\beta_{ik} + 2) \cdot (\alpha_{kj} + \beta_{kj} + 2) + 2 \cdot \alpha_{ik}}$$

otherwise α_j^{rec} and β_j^{rec} are set to zero.

4. For each neighbour j, node i computes:

$$\alpha_j = \alpha_{ij} + \alpha_j^{\text{rec}} \qquad \beta_j = \beta_{ij} + \beta_j^{\text{rec}}$$

5. For each neighbour j, if $\mathtt{E}(\text{Beta}(\alpha_j + 1, \beta_j + 1)) < tth$ then x_{ij} is discarded.
6. Node i computes its local value on the base of the non-discarded x_{ij}.

Notice that in order to preserve the nature of Aggregate Computing, each node computes locally and makes decisions deriving from the knowledge of its neighbourhood. The novelty is the application of a mechanism used in trust systems to monitor the neighbourhood and detect potential suspicious behaviours. The algorithm is parameterised by the two thresholds *maxError* and *tth*, which deserve empirical evaluation, as they characterise the attitude of the node to trust perturbed values and other nodes sharing perturbed values, respectively. Analogously, the recommendation mechanism represents another option that can be activated to take advantage of the knowledge shared by other (trusted) nodes. Finally, the generalisation to non-numeric field domains is possible without changing the nature of the approach and by adapting the semantics of the functions and operators used in the algorithm above.

Trust implementation in ScaFi In order to study the trust algorithm proposed in Section 7.6.1, we have applied it to the case of a gradient computation. Here, we describe the ScaFi implementation of the core trust logic in action, working as a high-level formal specification of the proposed solution, a ready-to-use JVM-based implementation, and as fully-reproducible simulation code.

The gradient algorithm presented in Section 7.2.2 can be extended to use trust as follows (new lines are highlighted):

```
trait BasicTrustMechanism extends TrustMechanism { self: AggregateProgram with
    Env with Lib =>
  case class BulkMetrics(s: Double, xmean: Double)
  case class TrustParams(a: Double, b: Double, numObservations: Int)
  case class TrustProfile(nbrId: ID, params: TrustParams)

  def bulkMetrics(value: Double): BulkMetrics = {
    def nbrVal = nbr { value }
    val n = countHood(nbrVal.isFinite)
    val sumValues = sumHood(mux(nbrVal.isFinite) { nbrVal } { 0.0 })
    val xmean = sumValues / n
    val sumSqDev = sumHood(mux(nbrVal.isFinite) { Math.pow(value - xmean, 2) }
    { 0.0 })
    val s = Math.sqrt(sumSqDev / n)
    BulkMetrics(s, xmean)
  }

  type MutableField[T] = MMap[ID,T]
  def MutableField[T](): MutableField[T] = MMap[ID,T]()
  type AlfaBetaPair = (Double, Double)
  type AlfaBetaHistory = List[AlfaBetaPair]

  def trustProfile(field: Double, bmetrics: BulkMetrics): TrustProfile = {
    val BulkMetrics(s: Double, xmean: Double) = bmetrics
    val (nbrId, nbrVal) = nbr{ (mid(), field) }
    val deviation = Math.abs(nbrVal - xmean)
    val maxError = Math.max(s, errorLB)

    val m = rep(MutableField[AlfaBetaHistory]()){ m => m }
    val history = m.getOrElse(nbrId, List())
    val obsEval = if(deviation > maxError) (0.0, 1.0) else (1.0, 0.0)
    m.put(nbrId, (obsEval :: history).take(observationWindow))

    val obss = m.getOrElse(nbrId, List())
    val (a,b) = obss.foldRight((0.0,0.0))((t,u) => (t._1+u._1, t._2+u._2))
    TrustProfile(nbrId, TrustParams(a, b, obss.size))
  }

  override def eval(m: Metrics, ifTrusted: Double, ifDistrusted: Double) = {
    val TrustParams(a, b, numObs) = trustParams(m)
    val trustVal = beta(a, b)
    val trusted = if (numObs >= minObservations) trustable(trustVal) else true
    val value = mux(trusted) { ifTrusted } { ifDistrusted }
    EvalResults(value, isTrusted)
  }

  def trustParams(m: Metrics): TrustParams // ABSTRACT

  def beta(a: Double, b: Double) = (a+1)/(a+b+2)
  def trustable(trustValue: Double): Boolean = trustValue >= trustThreshold
}
```

Figure 7.16: ScaFi scaffolding for trust mechanisms based on beta distribution.

```
class PlainTrust extends BasicTrustMechanism { self: AggregateProgram with Env
    with Lib =>
  override type Metrics = TrustMetrics
  case class TrustMetrics(bmetrics: BulkMetrics, trustProfiles: Map[ID,
    TrustProfile])

  override def trustParams(m: TrustMetrics): TrustParams =
    m.trustProfiles(nbr{mid}).params

  override def metrics(value: Double): TrustMetrics = {
    val bmetrics = bulkMetrics(value)
    TrustMetrics(bmetrics, mapHood{ trustProfile(value, bmetrics) } )
  }
}
```

Figure 7.17: Implementation, in ScaFi, of the plain trust algorithms.

```
def gradient(source: Boolean, trust: TrustMechanism): Double =
  rep(Double.PositiveInfinity){ distance =>
    mux(source) { 0.0 } {
      val trustMetrics = trust.metrics(distance)

      foldhoodPlus(Double.PositiveInfinity)(Math.min)(
        trust.eval(trustMetrics,
          whenTrusted = nbr { distance } + nbrRange,
          whenDistrusted = Double.PositiveInfinity
        ).value)
    }
  }
```

A trust mechanism can be thought of as a building block to be invoked at particular stages of the gradient computation, and whose interface (expressed as Scala trait) can be of the kind:

```
trait TrustMechanism {
  type Metrics
  case class EvalResults(value: Double, trusted: Boolean)

  def metrics(value: Double): Metrics
  def eval(metrics: Metrics, whenTrusted: Double, whenDistrusted: Double):
    EvalResults
}
```

In particular, two main phases of the gradient algorithm can be recognised:

1. *Data collection* – when the contributions of the neighbours are retrieved: at

```
class TrustWithRecomms extends BasicTrustMechanism { self: AggregateProgram
    with Env with Lib =>
  override type Metrics = RecommMetrics
  case class RecommMetrics(bmetrics: BulkMetrics,
                           nbrTrustProfiles: Map[ID,Map[ID,TrustProfile]])

  override def metrics(field: Double): RecommMetrics = {
    val bmetrics = bulkMetrics(field)
    val localTrustProfiles: Map[ID, TrustProfile] = mapHood { trustProfile(
    field, bmetrics) }
    RecommMetrics(bmetrics, mapHood{ nbr(localTrustProfiles) })
  }

  override def trustParams(m: RecommMetrics): TrustParams = {
    def localTrustProfiles = m.nbrTrustProfiles(mid)
    val nbrId = nbr { mid() }
    val (aRec: Double, bRec: Double) = m.nbrTrustProfiles
      .mapValues(_.get(nbrId).map(p => (p.params.a, p.params.b))
                            .getOrElse(0.0, 0.0))
      .foldLeft((0.0, 0.0))((acc, value) => {
        // i = mid, j = nbrId ; a_j and b_j calculated from all nbrs k != i,j
        val TrustParams(a_ik, b_ik, _) =
          localTrustProfiles.get(value._1).map(_.params)
                            .getOrElse(TrustParams(0.0, 0.0, 0))
        val (a_kj, b_kj) = (value._2._1, value._2._2)
        val denom = (b_ik + 2) * (a_kj + b_kj + 2) + 2 * a_ik
        (acc._1 + 2 * a_ik * a_kj / denom, acc._2 + 2 * a_ik * b_kj / denom)
      })
    val localParams = localTrustProfiles.get(nbrId)
                          .map(_.params).getOrElse(TrustParams(0.0, 0.0, 0))
    TrustParams(localParams.a + aRec, localParams.b + bRec, localTrustProfiles.
    size)
  }
}
```

Figure 7.18: Implementation, in ScaFi, of the recommendations-based trust algorithms.

this point, the "metrics" for trust evaluation can be computed;

2. *Minimisation* – when the contributions of the neighbours are used to deduce the new gradient value by applying the triangle inequality constraint: here is when trust can be applied in order to filter out the contributions from distrusted neighbours.

Trait `BasicTrustMechanism` (Figure 7.16) partially implements the `TrustMechanism` contract to provide the scaffolding of a trust mechanism that uses the mean square deviation of some value to derive trust profiles and an evaluation strategy based on the beta distribution. The concrete implementations of such basic trust mechanism, with and without recommendations (class `PlainTrust` and `TrustWithRecomms`, resp.), are shown in Figure 7.18. The idea behind the code design is the following: a first step is to collect as much context information as possible from the neighbourhood in order to elicit a reference system (`bulkMetrics`); then, for each neighbour, a trust profile is delineated (`trustProfile`) by "reading" the individual contribution against the bulk metrics; after that, a set of trust parameters for a neighbour profile is computed (`trustParams`) and used to calculate the trust score (`beta`), i.e., the trust field; finally, the trust score is checked against a threshold to choose whether or not the currently examined neighbour has to be trusted (`trustable`), resulting in the choice of `whenTrusted` or `whenDistrusted` values for the `EvalResults` to be returned. The code makes use of some utility functions: `countHood` counts for how many neighbours the given predicate is true; `sumHood` sums the neighbours' values for the provided expression; and `mapHood` returns a map from neighbours' `ID`s to the corresponding values of the provided expression.

The key differences between `PlainTrust` and `TrustWithRecomms` lie in how the trust metrics and parameters are computed. For the latter, notice how the `localTrustProfiles` are gathered from the neighbours.

It is worth noting that, though correctly implementing the desired approach, the presented solution has a drawback: it requires the definition of a *new* gradient algorithm that is aware of, or depends on, the provided `TrustMechanism`; ideally, there should be a way to inject the logic of trust into an existing algorithm, as a sort of orthogonal component. This can be considered as an interesting future work. A potential approach would be to work at the value-tree level of aggregate programs, by defining injection points for inputs and outputs of the trust component, in a

way similar to aspect-oriented programming.

Attack-resistant gradients for attack-resistant channels The channel algorithm is a reusable building block for computing, in a distributed way, a self-healing path from a source to a destination area. A channel of width `w` from `a` to `b` that leverages gradient function `g` can be implemented in ScaFi as follows:

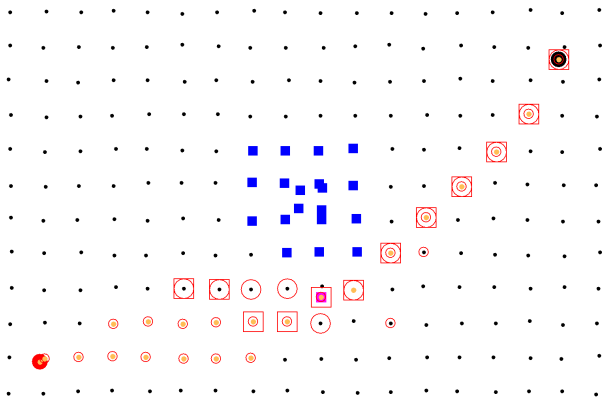
```
def channel(a: Boolean, b: Boolean, w: Double, g: Boolean => Double):
  Boolean =
  g(a) + g(b) <=  distBetween(a, b, g) + w

def distBetween(src: Boolean, dest: Boolean, g: Boolean => Double): Double
  = broadcast(src, g(dest))

def broadcast[V:Bounded](src: Boolean, field: V): V
  = G[V](src, field, v => v, nbrRange)
```

In this case study, the goal is to evaluate how the gradient algorithm under attack is able to support the formation of a correct channel. In practice, the following channel fields are computed:

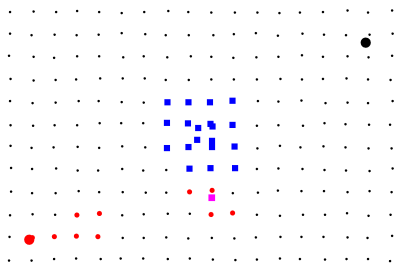
```
val cIdeal   = channel(isSrc, isTarget, width, gradient(_))
val cFake    = channel(isSrc, isTarget, width, gradient(_, fake=true))
val cTrust   = channel(isSrc, isTarget, width,
                       gradient(_, fake=true, trust=PlainTrust))
val cRecomms = channel(isSrc, isTarget, width,
                       gradient(_, fake=true, trust=Recommendations))
val (errFake, errTrust, errRecomms) = (cIdeal ^ cFake, cIdeal ^ cTrust, cIdeal
    ^ cRecomms)
```

and the error in the presence of fake contributions is measured by counting, with respect to the "right channel" in which no fakes are activated (`cIdeal`), the number of nodes with inverted boolean values (i.e., the number of `true` nodes in the XOR field), for three cases: no trust (`cFake`), plain trust (`cTrust`), and trust with recommendations (`cRecomms`). The idea is that, while the fake is able – in the absence of any trust mechanisms at work – to corrupt the gradient fields beneath the channel, effectively compromising the path to a level of complete uselessness, the adoption of trust-based gradient algorithms can provide enough resiliency to neutralise the perceived effect of the attacks. Figure 7.19 provides a pictorial representation of the simulation scenario, the evaluation approach and the expected outcomes.

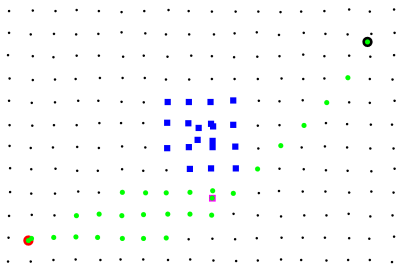
The result of the experiment for (a subset of) different runs is reported in



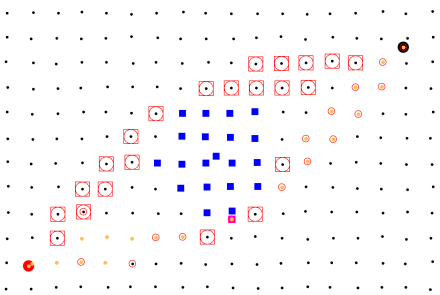
(a) Snapshot of the channel simulation at the initial stages. The big red and black nodes at the corners are the source and target nodes, respectively. Blue square nodes are obstacles. The magenta square node is the fake, which propagates random values, potentially distorting the gradient fields underlying the channel computation. Orange nodes belong to the ideal channel (i.e., the channel computed as if there were no fakes). The mere effect of the fake on the ideal channel is shown by the small red circles, which denote those nodes yielding a wrong value of channel membership; instead, the big red circles (resp., squares) are used to highlight errors committed while also *using* trust (resp., recommendations).



(b) The channel created by the fake never stabilises. This snapshot shows the channel path without using trust is completely compromised.



(c) Using trust, it is possible to preserve the channel. Notice that it might be slightly different from the ideal channel visible in (a).



(d) Sometimes, the reduction of error provided by trust in the gradients beneath the channel algorithm may result in a different path being chosen. Though reasonable, it still counts as an error in our evaluation.

Figure 7.19: Snapshots from the channel simulation.

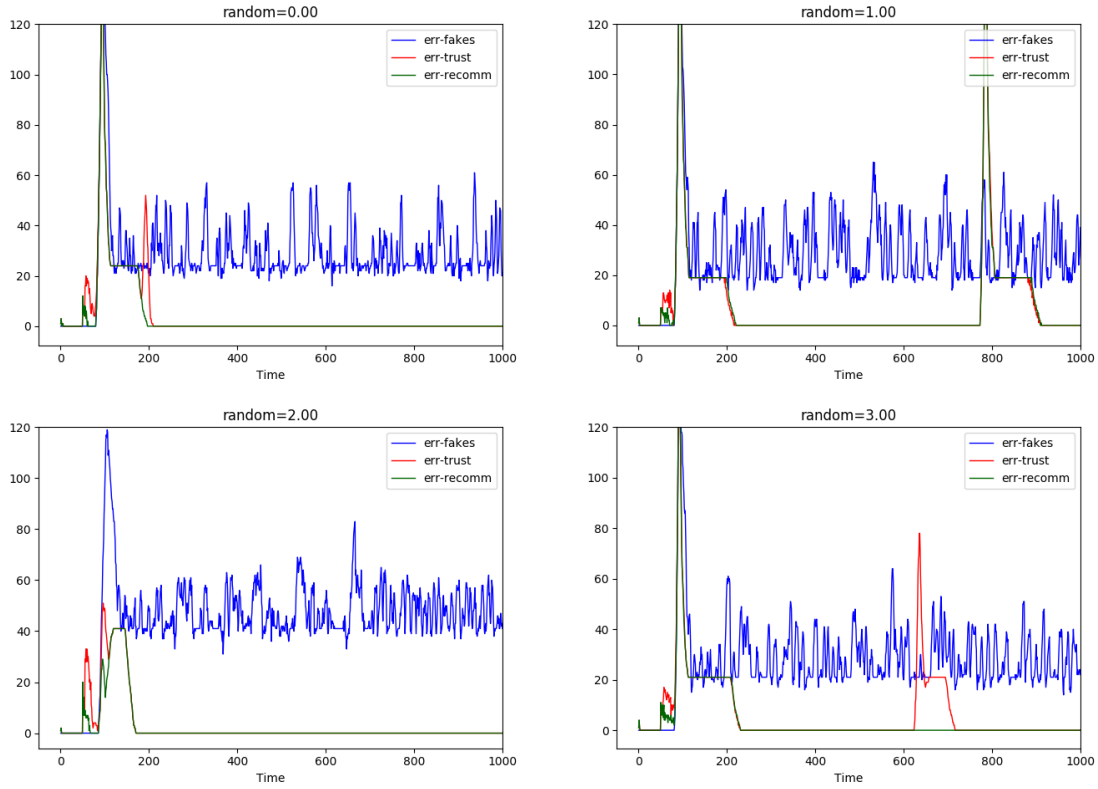


Figure 7.20: Each graph shows the evolution of the error across time for different random seeds. Configuration: fake appearing at $t = 80$; trust algorithm starting at $t = 50$; $trustThreshold = 0.90$; $errorLB = 8.0$.

Figure 7.20: when using trust, convergence is basically achieved, with the exception of some sporadic reconstructions of the channel where a noticeable error is registered during the transitory phase. Notice that, in general, the gradient and channel algorithms are self-stabilising; however, the fake node acts as a source of non-constant input and continuously perturbs them. The same experiments, launched for 2000 seconds, with $tth = 0.94$ and $errorLB = 8.0$, have shown that recommendations, after the initial error peak, maintain convergence (i.e., correct channel with null error) all the time in 17 out of 20 runs, where the remaining 3 runs only present one or two peaks of error that are quickly fixed, still exhibiting the correct channel for the majority of time.

In summary, the contribution is clear: gradient implementations adopting trust mechanisms improve stability and provide resistance to attacks in such a way that

they do not impact on higher level building blocks (and in turn to applications).

It is important to notice that attacking the gradients underlying the channel is the way by which an attacker can impact the system the most: in fact, by trying to provoke a disruptive global effect out of merely local contributions, it is possible to completely compromise the channel. In addition, some mechanism should be used to prevent a malevolent node from pretending to be a source or target. Instead, attacks directed at top-level channel values, while skipping the defence line provided by the trust-based gradient implementation, must hijack several nodes to produce a significant system-wide effect (e.g., a channel partition). Functionality might still be locally compromised, though. This aspect, which is left as an interesting future work, might possibly be tackled by enforcing invariants between different parts of an aggregate computation.

7.7 Final Remarks

This chapter covers theory and practice of ScaFi. ScaFi provides a different programming experience with respect to Protelis, as well as a different framework for language development. From a formal standpoint, the corresponding FScaFi calculus has a different expressiveness with respect to HFC, though they share a common core that exhibit the most useful properties. In ScaFi, the full power of the Scala programming language is at hand, even though attention should be paid when using certain features that may alter the execution flow—static checks could be implemented (e.g., as Scala compiler plugins) to intercept common mistakes. In addition to the DSL, ScaFi provides basic simulation facilities (additionally, it is also integrated with Alchemist for advanced simulation needs), a library of building blocks (also with support for aggregate processes—see Chapter 8), as well as an actor-based middleware for building aggregate-like distributed systems (see Chapter 9).

References

[Akk] `http://akka.io`. Retrieved October 15-th 2018. 2018.

[Apaa] `https://kafka.apache.org`. Retrieved October 15-th 2018. 2018.

[Apab] https://spark.apache.org. Retrieved October 15-th 2018. 2018.

[Aud+16] Giorgio Audrito, Ferruccio Damiani, Mirko Viroli, and Roberto Casadei. "Run-Time Management of Computation Domains in Field Calculus". In: *Foundations and Applications of Self* Systems, IEEE International Workshops on*. IEEE. 2016, pp. 192–197.

[Aud+17] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. "Compositional Blocks for Optimal Self-Healing Gradients". In: *Self-Adaptive and Self-Organising Systems (SASO), IEEE International Conference on*. IEEE. 2017.

[Aud+18] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. "Space-time universality of field calculus". In: *International Conference on Coordination Languages and Models*. Springer. 2018, pp. 1–20.

[Aud+19] Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. "A Higher-Order Calculus of Computational Fields". In: *ACM Transactions on Computational Logic* 20.1 (2019), pp. 1–55. DOI: 10.1145/3285956.

[BB04] S. Buchegger and J.-Y. Le Boudec. "A Robust Reputation System for Peer-to-Peer and Mobile Ad-hoc Networks". In: *2nd Workshop on the Economics of Peer-to-Peer Systems*. P2PEcon. 2004.

[BB06] Jacob Beal and Jonathan Bachrach. "Infrastructure for Engineered Emergence in Sensor/Actuator Networks". In: *IEEE Intelligent Systems* 21 (2 2006), pp. 10–19. DOI: 10.1109/MIS.2006.29.

[Bea+08] Jacob Beal, Jonathan Bachrach, Dan Vickery, and Mark Tobenkin. "Fast self-healing gradients". In: *Proceedings of the 2008 ACM symposium on Applied computing*. ACM. 2008, pp. 1969–1975.

[Bet16] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd., 2016. ISBN: 9781786464965. URL: /files/https://www.packtpub.com/application-development/implementing-domain-specific-languages-xtext-and-xtend-second-edition.

[Cal+17] Ben Calus, Bob Reynders, Dominique Devriese, Job Noorman, and Frank Piessens. "FRP IoT Modules As a Scala DSL". In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS 2017. Vancouver, BC, Canada: ACM, 2017, pp. 15–20. ISBN: 978-1-4503-5515-5. DOI: 10.1145/3141858.3141861. URL: http://doi.acm.org/10.1145/3141858.3141861.

[Cas+19a] Roberto Casadei, Christos Tsigkanos, Mirko Viroli, and Schahram Dustdar. "Engineering Resilient Collaborative Edge-Enabled IoT". In: *2019 IEEE International Conference on Services Computing (SCC)*. 2019, pp. 36–45. DOI: 10.1109/SCC.2019.00019.

[Cas+19b] Roberto Casadei, Mirko Viroli, Giorgio Audrito, Danilo Pianini, and Ferruccio Damiani. "Aggregate Processes in Field Calculus". In: *Coordination Models and Languages*. Ed. by Hanne Riis Nielson and Emilio Tuosto. Cham: Springer International Publishing, 2019, pp. 200–217. ISBN: 978-3-030-22397-7.

[Cas+20] Roberto Casadei, Mirko Viroli, Giorgio Audrito, and Ferruccio Damiani. "Aggregate Programming in Scala with ScaFi". In: 2020. Submitted to a journal.

[CAV18] Roberto Casadei, Alessandro Aldini, and Mirko Viroli. "Towards Attack-Resistant Aggregate Computing Using Trust Mechanisms". In: *Science of Computer Programming* 167 (2018), pp. 114–137. DOI: 10.1016/j.scico.2018.07.006.

[CPV16] Roberto Casadei, Danilo Pianini, and Mirko Viroli. "Simulating large-scale aggregate MASs with alchemist and scala". In: *Computer Science and Information Systems (FedCSIS), 2016 Federated Conference on*. IEEE. 2016, pp. 1495–1504.

[CV18] Roberto Casadei and Mirko Viroli. "Programming Actor-Based Collective Adaptive Systems". In: *Programming with Actors: State-of-the-Art and Research Perspectives*. Vol. 10789. Lecture Notes in Computer Science. Springer, 2018, pp. 94–122. DOI: 10.1007/978-3-030-00302-9_4.

[CV19] Roberto Casadei and Mirko Viroli. "Coordinating Computation at the Edge: a Decentralized, Self-Organizing, Spatial Approach". In: *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*. 2019, pp. 60–67. DOI: 10.1109/FMEC.2019.8795355.

[Die+16] Felix Dietze, Johannes Karoff, André Calero Valdez, Martina Ziefle, Christoph Greven, and Ulrik Schroeder. "An Open-Source Object-Graph-Mapping Framework for Neo4j and Scala: Renesca". In: *Availability, Reliability, and Security in Information Systems*. Ed. by Francesco Buccafurri, Andreas Holzinger, Peter Kieseberg, A Min Tjoa, and Edgar Weippl. Cham: Springer International Publishing, 2016, pp. 204–218. ISBN: 978-3-319-45507-5.

[DM82] Luis Damas and Robin Milner. "Principal Type-schemes for Functional Programs". In: *Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: ACM, 1982, pp. 207–212. ISBN: 0-89791-065-6. DOI: 10.1145/582153.582176.

[Eff+12] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. "Xbase: implementing domain-specific languages for Java". In: *ACM SIGPLAN Notices*. Vol. 48. 3. ACM. 2012, pp. 112–121.

[GBS08] S. Ganeriwal, L. K. Balzano, and M. B. Srivastava. "Reputation-based Framework for High Integrity Sensor Networks". In: *ACM Trans. Sen. Netw.* 4.3 (2008), pp. 1–37.

[Gho11] Debasish Ghosh. "DSL for the uninitiated". In: *Commun. ACM* 54.7 (2011), pp. 44–50. DOI: 10.1145/1965724.1965740. URL: https://doi.org/10.1145/1965724.1965740.

[Han+14] G. Han, J. Jiang, L. Shu, J. Niu, and H.-C. Chao. "Management and Applications of Trust in Wireless Sensor Networks: A Survey". In: *Journal of Computer and System Sciences* 80.3 (2014). Special Issue on Wireless Network Intrusion, pp. 602–617.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: A Minimal Core Calculus for Java and GJ". In: *ACM Transactions on Programming Languages and Systems* 23.3 (2001), pp. 396–450.

[JI02] A. Jøsang and R. Ismail. "The beta reputation system". In: *15th Bled Conf. on Electronic Commerce*. 2002.

[LK87] Frank C. H. Lin and Robert M. Keller. "The Gradient Model Load Balancing Method". In: *IEEE Trans. Softw. Eng.* 13.1 (1987), pp. 32–38. ISSN: 0098-5589. DOI: http://dx.doi.org/10.1109/TSE.1987.232563.

[Mou+15] H. Mousa, S. Ben Mokhtar, O. Hasan, O. Younes, M. Hadhoud, and L. Brunie. "Trust Management and Reputation Systems in Mobile Participatory Sensing Applications: A Survey". In: *Computer Networks* 90 (2015), pp. 49–73. ISSN: 1389-1286.

[OR14] Martin Odersky and Tiark Rompf. "Unifying functional and object-oriented programming with scala". In: *Communications of the ACM* 57.4 (2014), pp. 76–86.

[OZ05] Martin Odersky and Matthias Zenger. "Scalable component abstractions". In: *ACM SIGPLAN Notices* 40.10 (Oct. 2005), p. 41. ISSN: 03621340. DOI: 10.1145/1103845.1094815.

[PKK16] B. Priayoheswari, K. Kulothungan, and A. Kannan. "Beta Reputation and Direct Trust Model for Secure Communication in Wireless Sensor Networks". In: *Int. Conf. on Informatics and Analytics*. ICIA-16. ACM, 2016, pp. 1–5.

[PVB15] Danilo Pianini, Mirko Viroli, and Jacob Beal. "Protelis: practical aggregate programming". In: *Symposium on Applied Computing*. ACM. 2015, pp. 1846–1853. DOI: 10.1145/2695664.2695913.

[VCP16] Mirko Viroli, Roberto Casadei, and Danilo Pianini. "On execution platforms for large-scale aggregate computing". In: *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. ACM. 2016, pp. 1321–1326.

[Vir+18] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. "Engineering Resilient Collective Adaptive Systems by Self-Stabilisation". In: *ACM Transaction on Modelling and Computer Simulation* 28.2 (2018), 16:1–16:28. ISSN: 1049-3301. DOI: 10.1145/3177774.

[Vli98] John M Vlissides. *Pattern hatching: design patterns applied*. Addison-Wesley Reading, 1998.

[Voe13] M. Voelter. *DSL Engineering: Designing, Implementing and Using Domain-specific Languages*. CreateSpace Independent Publishing Platform, 2013. ISBN: 9781481218580. URL: https://books.google.it/books?id=J2i0lwEACAAJ.

[Yu+12] Y. Yu, K. Li, W. Zhoub, and P. Lib. "Trust Mechanisms in Wireless Sensor Networks: Attack Analysis and Countermeasures". In: *Journal of Network and Computer Applications* 35.3 (2012), pp. 867–880.

Chapter 8

Dynamic Collective Computations with Aggregate Processes

The group process contains the secret of collective life [...]

Mary Parker Follett · *The New State: Group Organization the Solution of Popular Government*

Contents

In the basic field calculus framework, a program consists of a single computation, possibly organised in a static tree of sub-computations whose domain can change dynamically (through the `branch` construct). Though universal [Aud+18], the field calculus does not provide effective mechanisms for modelling *dynamic* field computations that may spring into existence, spread to involve a dynamic team of devices, and then retract and collapse once job is done. These kinds of "computational bubbles" are called *aggregate processes*. In this chapter, the aggregate process abstraction is described (Section 8.1), together with a field calculus extension (Section 8.2), a corresponding implementation in the ScaFi framework (Section 8.3), and an account of programming techniques (Section 8.4). Finally, aggregate processes are shown in action through two case studies of opportunistic messaging and drone reconnaissance (Section 8.5).

8.1 Aggregate Processes: Introduction

8.1.1 Motivation

Scenarios like the IoT, CPS, and pervasive computing bring about a future deeply interconnecting the digital and physical world, and where environments are smart, open and rich of heterogeneous devices providing services in a cooperative way. There, computational events might trigger multiple processes that are highly contextual and hence fundamentally related to their space-time situation, though possibly involving several computational devices and infrastructural layers (cf., edge, fog, and cloud). Openness and dynamism of deployment environments,

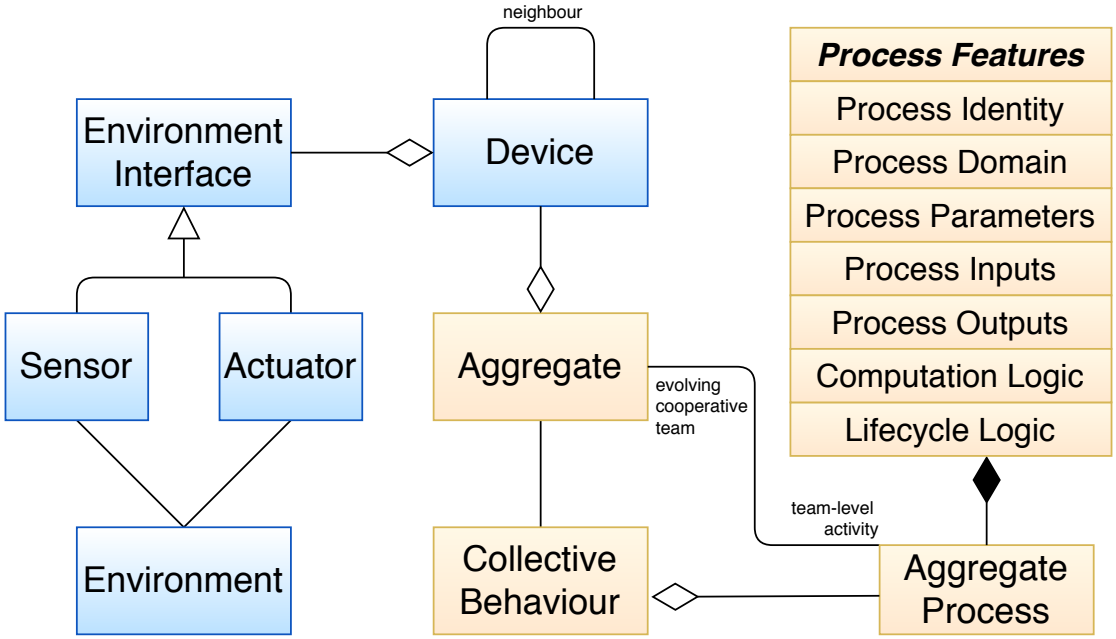


Figure 8.1: Logical UML model of IoT systems, comprising first-class aggregates cooperatively playing some aggregate behaviour which may include multiple concurrent aggregate processes. Colours are used to discriminate between individual (blue) and collective (orange) concepts.

then, require such processes to be dependable, self-adaptive and self-organising in order to maintain coherence and functionality across the unpredictable and inevitable context changes and adversary events, and to opportunistically activate and execute wherever and whenever their existence conditions hold—whether they are by-design or emergent. According to this vision, the notion of *aggregate processes* is proposed as key abstraction for modelling

> *dynamic, context-driven and collective activities that concurrently span and overlap into a (possibly mobile, large-scale) collection of situated, computational devices (which we call an* aggregate *or* ensemble*).*

Figure 8.1 describes the role of aggregate processes in IoT systems, in terms of relationships with typical entities involved (situated devices—i.e., the *things*) and new first-class citizens like aggregates (collectives of things).

8.1.2 Requirements

In order to be more systematic in the characterisation of our aggregate process abstraction, we make explicit a set of requirements or desiderata which are based on the aforementioned vision and on pragmatic aspects of the software engineering practice:

- *aggregate stance* — to promote pervasive adaptation, aggregate processes should abstract the individual device and seamlessly regulate the behaviour of an ensemble across scales, density, and heterogeneity;
- *dynamicity and context-orientation*: aggregate processes should conveniently support the implementation of dynamic, distributed, spatio-temporal activities where context plays a major role and continuous change is the norm;
- *opportunistic resource exploitation*: aggregate processes should support dynamic and opportunistic execution across the heterogeneous devices spread in the physical and computational environment;
- *intrinsic resiliency* — aggregate process implementations should provide formal guarantees about independence to large classes of environmental dynamics and faults;
- *conceptual, methodological, and technological integration* — aggregate processes should integrate with mainstream development paradigms, techniques and tools.

8.1.3 Features of aggregate processes

Multiple aspects and perspectives – structural, behavioural, and interactional – need to be considered to fully characterise an aggregate process. First, terminologically, we shall use the term *process* to refer to both process *types* and process *instances*, i.e., concrete living instantiations of process types. Then, in practice, when specifying a process, a designer should be able to control the following aspects:

- *process generation* — where and when a process is spawned, by whom, and with which construction parameters;

- *process destruction* — the events triggering shutdown and the desired tear-down dynamics;
- *process identity* — how to distinguish between different instances of the same process;
- *process domain or shape* — how the set of devices running a process changes across space and time through "extension" and "shrinking" of the process boundary;
- *process parameters* — how other processes and data can be used as construction-time and run-time parameters;
- *process logic* — what actual collective computation must be carried on by the process; and
- *inter-process interaction* — the mechanisms that process instances can use to interact with one another.

8.2 Formalisation

8.2.1 On "multiple alignments"

Conceptually, and technically, FC is used to specify a "single field computation" working on the entire available domain. As a paradigmatic example, consider a `gradient` [Vir+18; Aud+17; LLM17], namely, a field of hop-by-hop distances based on local estimates `metric` (a field of neighbouring real values) from the closest node in *source* (a field of boolean values):

```
def gradient(source, metric) {
  rep(infinity)(distance =>
     mux { source } { 0 } { minHoodPlus( nbr{distance} + metric ) }
  )
}

def limitedGradient(source, metric, area) {
  branch { area } { gradient(source, metric) } { infinity }
}
```

If `sns` is a sensor giving `true` only at a device s (and `false` everywhere else) and `nbrRange` is a sensor giving local estimate distances from neighbours (as a range de-

Syntax:

$\texttt{P} ::= \overline{\texttt{F}}\ \texttt{e}$	program	$\texttt{F} ::= \texttt{def d}(\overline{\texttt{x}})\ \{\texttt{e}\}$	function decl.
$\texttt{e} ::= \texttt{x} \mid \texttt{v} \mid (\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{e} \mid \texttt{e}(\overline{\texttt{e}}) \mid \texttt{rep(e)\{(x) => e\}} \mid \texttt{nbr\{e\}} \mid \texttt{spawn(e,e,e)}$	expr.		
$\texttt{v} ::= \phi \mid \ell$	value	$\phi ::= \overline{\delta} \mapsto \overline{\ell}$	nbr field value
$\ell ::= \texttt{f} \mid \texttt{c}(\overline{\ell})$	local value	$\texttt{f} ::= \texttt{b} \mid \texttt{d} \mid (\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{e}$	function value

Value-trees and value-tree environments:

$\theta ::= \texttt{v} \mid \texttt{v}\langle\overline{\theta}\rangle \mid \overline{\texttt{v}} \mapsto \overline{\theta}$	value-tree	$\Theta ::= \overline{\delta} \mapsto \overline{\theta}$	value-tree env.

Auxiliary functions:

$args((\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{e}) = \overline{\texttt{x}}$	$body((\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{e}) = \texttt{e}$	$name((\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{e}) = \tau$
$args(\texttt{d}) = \overline{\texttt{x}}$ if $\texttt{def d}(\overline{\texttt{x}})\{\texttt{e}\}$	$body(\texttt{d}) = \texttt{e}$ if $\texttt{def d}(\overline{\texttt{x}})\{\texttt{e}\}$	$name(\texttt{d}) = \texttt{d}$
$\rho(\texttt{v}\langle\overline{\theta}\rangle) = \texttt{v}$		$name(\texttt{b}) = \texttt{b}$
$\pi_i(\texttt{v}\langle\theta_1,\ldots,\theta_n\rangle) = \theta_i$	if $1 \leq i \leq n$	else $\bullet$
$\pi^{\texttt{f}}(\texttt{v}\langle\theta_1,\ldots,\theta_n\rangle) = \theta_n$	if $name(\rho(\theta_1)) = name(\texttt{f})$	else $\bullet$
$\pi^k(\overline{\texttt{v}} \mapsto \overline{\theta}) = \theta_i$	s.t. $\texttt{v}_i = k$ if it exists	else $\bullet$
$F(\theta) = \texttt{v}\langle\overline{\theta}\rangle$	if $\theta = \texttt{pair(v, True)}\langle\overline{\theta}\rangle$	else $\bullet$

$$\forall aux \in \rho, \pi_i, \pi^{\texttt{f}}, \pi^k, F: \begin{cases} aux(\bullet) = \bullet & \\ aux(\delta \mapsto \theta, \Theta) = aux(\Theta) & \text{if } aux(\theta) = \bullet \\ aux(\delta \mapsto \theta, \Theta) = \delta \mapsto aux(\theta), aux(\Theta) & \text{if } aux(\theta) \neq \bullet \end{cases}$$

Rules for expression evaluation: $\boxed{\delta; \Theta; \sigma \vdash \texttt{e} \Downarrow \theta}$

$$\text{[E-APP]}\ \frac{\delta; \pi_1(\Theta); \sigma \vdash \texttt{e} \Downarrow \theta \quad \delta; \pi_{i+1}(\Theta); \sigma \vdash \texttt{e}_i \Downarrow \theta_i \text{ for } i = 1 \ldots n \quad \texttt{f} = \rho(\theta) \quad \theta' = (\!|\texttt{f}|\!)^{\pi^{\texttt{f}}(\Theta)}_{\delta,\sigma}(\rho(\overline{\theta})) \text{ if } \texttt{f} : \texttt{b}\ else\ \delta; \pi^{\texttt{f}}(\Theta); \sigma \vdash body(\texttt{f})[args(\texttt{f}) := \rho(\overline{\theta})] \Downarrow \theta'}{\delta; \Theta; \sigma \vdash \texttt{e}(\overline{\texttt{e}}) \Downarrow \rho(\theta')\langle\theta, \overline{\theta}, \theta'\rangle}$$

$$\frac{\text{[E-LOC]}}{\delta; \Theta; \sigma \vdash \ell \Downarrow \ell\langle\rangle} \qquad \frac{\text{[E-FLD]}\ \phi' = \phi|_{\mathcal{D}_\Theta \cup \{\delta\}}}{\delta; \Theta; \sigma \vdash \phi \Downarrow \phi'\langle\rangle}$$

$$\frac{\text{[E-NBR]}\quad \Theta_1 = \pi_1(\Theta) \quad \delta; \Theta_1; \sigma \vdash \texttt{e} \Downarrow \theta_1}{\delta; \Theta; \sigma \vdash \texttt{nbr\{e\}} \Downarrow \rho(\Theta_1)[\delta \mapsto \rho(\theta_1)]\langle\theta_1\rangle}$$

$$\text{[E-REP]}\ \frac{\begin{array}{l}\delta; \pi_1(\Theta); \sigma \vdash \texttt{e}_1 \Downarrow \theta_1 \\ \delta; \pi_2(\Theta); \sigma \vdash \texttt{e}_2[\texttt{x} := \ell_0] \Downarrow \theta_2 \qquad \ell_0 = \begin{cases} \rho(\pi_2(\Theta))(\delta) & \text{if } \delta \in \mathcal{D}_\Theta \\ \ell_1 & \text{otherwise} \end{cases} \\ \ell_1 = \rho(\theta_1) \quad \ell_2 = \rho(\theta_2)\end{array}}{\delta; \Theta; \sigma \vdash \texttt{rep}(\texttt{e}_1)\{(\texttt{x}) \texttt{=>} \texttt{e}_2\} \Downarrow \ell_2\langle\theta_1, \theta_2\rangle}$$

$$\text{[E-SPAWN]}\ \frac{\begin{array}{ll}\delta; \pi_i(\Theta); \sigma \vdash \texttt{e}_i \Downarrow \theta^i & \text{for } i \in 1, 2, 3 \\ k_1, \ldots, k_n = \rho(\theta^2) \cup \bigcup\{\mathcal{D}_{\pi_4(\Theta(\delta'))} & \text{for } \delta' \in \mathcal{D}_\Theta\} \\ \delta; \pi^{k_i}(\pi_4(\Theta)); \sigma \vdash \rho(\theta_1)(k_i, \rho(\theta_3)) \Downarrow \theta_i & \text{for } i \in 1, \ldots, n\end{array}}{\delta; \Theta; \sigma \vdash \texttt{spawn}(\texttt{e}_1, \texttt{e}_2, \texttt{e}_3) \Downarrow F(\overline{k} \mapsto \rho(\overline{\theta}))\langle\theta^1, \theta^2, \theta^3, F(\overline{k} \mapsto \overline{\theta})\rangle}$$

Figure 8.2: Syntax and device semantics for the FC (extended part in grey).

tector would support), then the main expression `gradient(sns,nbrRange)` gives a field stabilising to a situation where each device is mapped to its (hop-by-hop, nearest) distance to s [Aud+19; Vir+18; Aud+17; LLM17; DV15]. If multiple devices are sources, estimated distance considers the nearest source.

There are mechanisms in FC to tweak this "single field computation" model. First of all, one could realise two computations by a field of pairs of values, say `(v1,v2)`: e.g., expression `(gradient(sns1,nbrRange), gradient(sns2,nbrRange))` would actually generate two completely independent gradient computations. The same approach is applicable to realise an arbitrary number of computations, but this practically works only if the number of such computations is small, known, and uniform across space and time, otherwise, FC has no mechanism to capture the abstraction of "aligned iteration" over a collection of values conceptually belonging to different computations.

A second key aspect involves the ability to restrict the domain of a computation. It is true that, by branching, one can prevent evaluation of some subexpressions—e.g., in function `limitedGradient`, if `area` is a boolean field giving `true` to a small subdomain, then computation of `gradient` is limited there. However, this approach has limitations as well: if one wants to limit a gradient to span the ball-like area where distances from the source are smaller than a given value, hence setting `area` to "`gradient(source,metric) < range`", there would be no direct way of avoiding computation of `gradient` outside that limited ball, because the decision on whether an event is inside or outside the ball has to be reconsidered everywhere and everytime.

So, technically, in FC there are no constructs to directly model, e.g., a reusable function that turns a field of boolean `sources` into a collection of independent gradients, one per source: that would require to create a field of lists of reals, of arbitrary size across space-time, but crucially this would not correctly support alignment. More generally, and although being universal [Aud+18], FC falls short in expressing the situation in which a field computation is composed of a set of subcomputations that is dynamic in the sense that has changing size over space and time. But this is precisely what is needed to support aggregate processes.

8.2.2 The `spawn` Construct Extension

We formalise our notion of *aggregate process* by extending FC with a `spawn` mechanism essentially carrying on a multiple aligned execution of concurrent computations, managing their life-cycle (i.e., activation, execution, disposal). Figure 8.2 (first frame) presents the syntax and device semantics of FC, where the grey-boxed parts correspond to the new `spawn` construct. Syntactically, a collection of aggregate process is modelled by a `spawn`-*expression* $\texttt{spawn}(\texttt{e}_b, \texttt{e}_k, \texttt{e}_i)$. Expression $\texttt{e}_b$ captures process behaviour: it is a function (of informal type $k \to a \to \langle v, bool \rangle$) taking a process key (i.e., an identifier) and an input argument, and returning a pair of an output value and a boolean stating whether the process should be maintained alive or not. Expression $\texttt{e}_k$ defines a field of process keysets to add at each location (device); and $\texttt{e}_i$ is the input field to feed processes. The result of `spawn` is a field of maps from process keys to values. As a result, we can precisely define an *aggregate process with key* k as the projection to k of the field of maps resulting from `spawn`, that is, the computational field associating each event to the value corresponding to k at that event—as this may simply be absent at an event, aggregate processes are to be considered partial fields over the whole domain.

The semantic details of `spawn` are presented in grey in Figure 8.2. On the second frame, we allow the expression of vtrees also as $\overline{\texttt{v}} \mapsto \overline{\theta}$, i.e., as a map from keys to vtrees. On the third frame, we define auxiliary functions ρ, π_i, π^k for extracting from a vtree respectively: its root value, an ordered subtree by its index i, and an unordered subtree by its key k. It also defines a *filtering* function F which selects vtrees whose root is a pair $\texttt{pair}(\texttt{v}, \texttt{True})$, collapsing the root into `v`. All of these functions can be extended to maps (see *aux*), which are intended to be unordered vtree nodes for F, and vtree environments for ρ, π_i and π^k.

Finally, in fourth frame, we define the behaviour of construct `spawn`, formalised by the big-step operational semantics rule [E-SPAWN]: the sub-expressions $\texttt{e}_1$, $\texttt{e}_2$ and $\texttt{e}_3$ are evaluated and their results stored in vtrees θ^1, θ^2, θ^3 forming the first branches of the final result. Then, a list of *process keys* $\overline{k}$ is computed by adjoining *(i)* the keys currently present in the result $\rho(\theta^2)$ of $\texttt{e}_2$; *(ii)* the keys that any neighbour δ' broadcast in their last unordered sub-branch $\pi_4(\Theta(\delta'))$. To realise "multiple alignment", for each key k_i, the process $\rho(\theta^1)$ resulting from evaluation

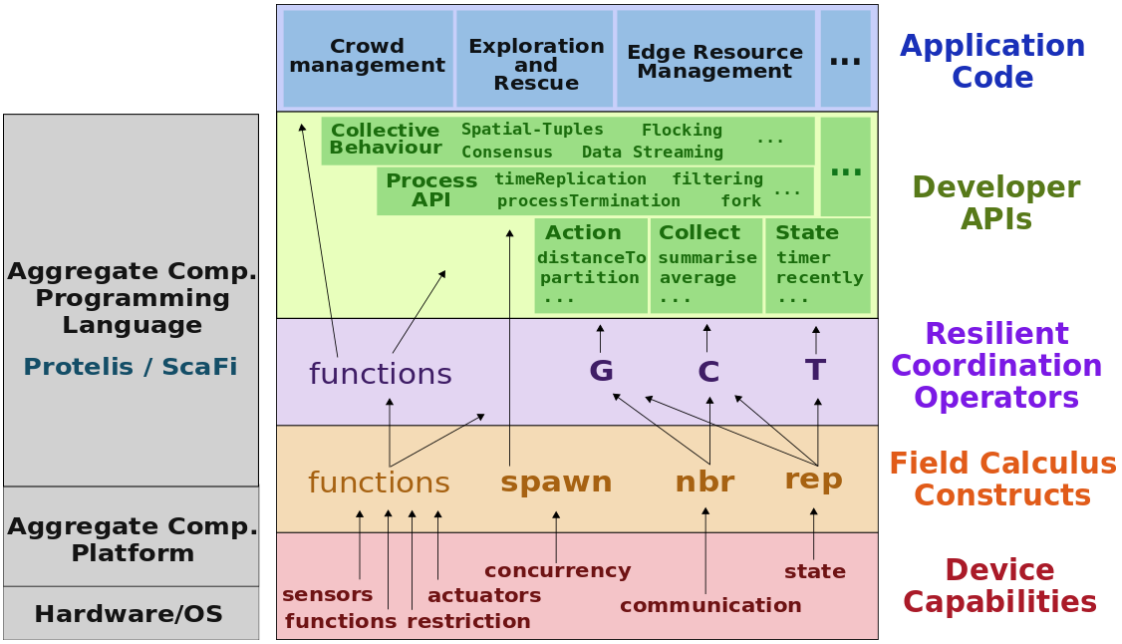


Figure 8.3: Aggregate computing engineering stack—the aggregate process concept, captured by the `spawn` construct, is the framework extension, discussed in this chapter, that opens to the dimension of concurrency.

of $\mathtt{e}_1$ is applied to k_i and the result $\rho(\theta^3)$ of $\mathtt{e}_3$, producing θ_i as a result. The map $\overline{k} \mapsto \overline{\theta}$ is then filtered by F, discarding evaluations resulting in a $\mathtt{pair}(\mathtt{v}, \mathtt{False})$, before being made available to neighbours. The same results $F(\overline{k} \mapsto \rho(\overline{\theta}))$ are also returned as the root of the resulting vtree.

8.3 Aggregate Process Implementation in ScaFi

In this section, we start from the technical issues pointed out in Section 8.2.1, which motivate the introduction of a construct `align` to handle arbitrary key-based alignment (Section 8.3.1), then we show the `spawn` construct in ScaFi (Section 8.3.2) – which extends the aggregate computing stack as per Figure 8.3 – before delving into its implementation details (Section 8.3.3).

8.3.1 Alignment and dynamic field expressions: the `align` construct

Results of field computations, at runtime, can be represented by hierarchical structures known as *value-trees (vtrees)*. A vtree is an ordered tree of values tracking the result of any evaluated expression. The operational semantics of the field calculus leverages vtrees; in other words, an evaluation of field expressions is a process building a vtree. Fundamental to the machinery and compositionality of the approach is the notion of *alignment* [Aud+16], by which evaluation, i.e., construction of vtrees, is defined in terms of other structurally-equivalent vtrees: the vtree corresponding to the previous round of the executing device, and the vtrees of neighbours devices. When two vtrees are not structurally-equivalent (i.e., they have different nodes), they do not "align", and hence one cannot be used with the other; notice, however, that two vtrees may *partially* align, and hence interaction is possible only within the aligned subtrees.

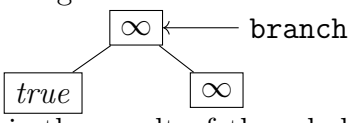
Indeed, when discussing the aggregate computing execution model (e.g., in Sections 5.1 and 7.3.3), we said that the local execution of a field computation yields an *export* message which is meant to be sent to neighbours in order to sustain the global behaviour of the aggregate. Such an export can also be seen as the state- and communication-related part of vtrees.

For instance, the expression

```
branch(isObstacle){ Double.PositiveInfinity }{ gradient(isSource) }
```

yields the following vtrees.

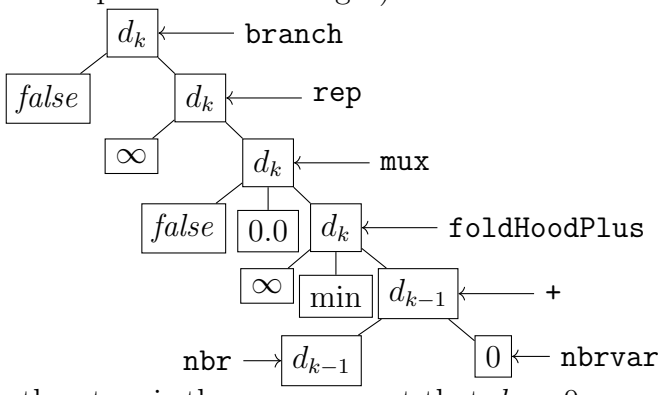
- For obstacle nodes, the following vtree:

  The root of the vtree (∞) is the result of the whole expression. The left sub-tree is the result of the first sub-expression (`isObstacle` which evaluates to *true*). Finally, the right sub-tree is the result of the branch taken (`Double.PositiveInfinity` which evaluates to ∞).

- For non-obstacle, non-source nodes, the following vtree (where d_k is the current distance estimate, d_{k-1} the previous one, and the main sub-expressions

relative to nodes are reported on their right):



For source nodes the vtree is the same, except that $d_k = 0$.

Notice that both state and communication are based on alignment: `rep` retrieves the value to work on from the previous vtree of the device, and `nbr` gets the values from neighbours by observing the nodes in the corresponding vtrees that have the same place in the computation as the current vtree node. In other words, interaction works on a structural basis where order matters.

However, dynamicity – due to potentially different and unknown activities – would break ordering, possibly leading to ambiguous or inconsistent vtree entries.

Paradigmatic example of the issue: limited multi-gradient The `gradient` function already supports the creation of a gradient from multiple sources: it is sufficient to build a source input field that is true in correspondence of multiple nodes.

```
val isSource = sense[Boolean]("source")
gradient(isSource)
```

With this approach, however, the resulting gradient field could not be used, e.g., to collect information into a source from nodes that are beyond the midpoint between that source and another adjacent source. The solution is to leverage multiple independent gradient computations that can overlap within the aggregate. For a fixed number of gradients, the following code works.

```
(gradient(source1), gradient(source2), ...)
```

However, there is no trivial way to handle a dynamic number of gradients (that are to be generated and destroyed as new sources activate or deactivate, resp.) without breaking alignment. For instance, the following code

```
val sources: Set[ID] = // gossip the source set
val gradients: Map[ID,Double] =
  sources.map(source => source -> gradient(source==mid)).toMap
```

would break in unexpected ways.

Solution: alignment over arbitrary keys The previous example involves evaluating a field expression in an iterative context. When mapping a dynamic collection, the number of elements and the order of traversal do not allow for drawing a consistent correspondence between two iteration steps of two devices, unless one manually introduce keys or tags for the field expressions, e.g., based the identity of the mapped elements. Therefore, we address this problem by a *new primitive mechanism*, called `align`, with signature

```
def align[K,V](key: K)(proc: K => V): V
```

to enable alignment on arbitrary keys, namely, to introduce a new computation scope by inserting a vtree node tagged with the provided key. Upon this, the previous code can be fixed as follows:

```
val gradients: Map[ID,Double] =
  sources.map(source =>
    source -> align(source){ _ => gradient(source==mid) }
  ).toMap
```

However, this approach is quite low-level, and does not properly handle lifecycle (currently expressed by field `sources` by means of gossiping). Therefore, we use the principle explained in this section to define a more expressive construct that provides both aligned execution and automatic propagation of keys. Such construct, `spawn`, effectively provides an implementation of our aggregate process abstraction.

8.3.2 Aggregate processes in ScaFi

The `spawn` primitive supports our notion of aggregate processes by handling activation, propagation, merging, and disposal of process instances (for a specified kind of process). Coherently with the formalisation in Section 8.2, it has signature:

```
def spawn[K,A,R](process: K => A => (R,Boolean),
                 newKeys: Set[K],
                 args: A): Map[K,R]
```

It is a generic function, parametrised by three types:

1. *`K`* — the type of process *keys*;
2. *`A`* — the type of process *arguments* (or inputs);
3. *`R`* — the type of process *result*.

The function accepts three formal parameters:

1. `process` — has type `K=>A=>(R,Boolean)` and expresses the computation logic of the process by a curried function taking a key, an argument, and then returning a pair of the computation result and a boolean *status* value expressing whether the current device is willing to participate in the process instance or not;
2. `newKeys` — is the set of keys of the processes to be spawned; and
3. `args` — is the "runtime argument" for the process instances active in this round.

Remember that values are fields—e.g., `newKeys` is a field of sets which may have entries only in specific devices and execution rounds, and `args` is a field whose values of type `A` may differ in different space-time locations. By a local perspective, `spawn` accepts a *set* of keys to allow *generation* of zero or more process instances at the device in the current round. Note that a process key has a twofold role: it works both as a *process identifier* (PID) and as *initialisation* or *construction parameter*. When different construction parameters should result in different process instances, it is sufficient to instantiate type `K` with a data structure type including both pieces of information and with proper equality semantics. Notice that if a new key already belongs to the set of active processes, there will be no actual generation

```
val vm: RoundVM = // provides access to virtual machine calls

def spawn[K, A, R](process: K => A => (R, Boolean),
                   newKeys: Set[K],
                   args: A
                  ): Map[A,R] = {
  rep(Map[K, R]()) { case processMap => {
    // 1. Take active process instances from my neighbours
    val nbrProcs = includingSelf.unionHoodSet(nbr{ processMap }.keySet)

    // 2. New processes to be spawn
    val newProcs = newKeys

    // 3. Get all processes to be executed, run them, and update their state
    (nbrProcs ++ newProcs)
      .map { p =>
        vm.newExportStack
        val result = align(puid) { _ => process(p)(args) }
        // Discard the export of the previous step if status is false
        if(result._2) vm.mergeExport else vm.discardExport
        p -> result
      }.collect { case(p,pi) if pi._2 => p -> pi._1 }.toMap
  } }
}
```

Figure 8.4: Simple implementation of `spawn` in ScaFi.

(or restart) but *merging* instead, since identity is the same as an existing process instance. Finally, note also that the outcome of `spawn` (a map from process keys to process result values) can in turn be used to fork other process instances or as input for other processes; i.e., the basic means for processes to interact is to connect the corresponding `spawn`s with data.

8.3.3 Behind-the-scenes: `spawn` implementation

To provide an intuition of the operational semantics of aggregate processes (formalised in Section 8.2), we take a look at the implementation of `spawn`, illustrated in the listing of Figure 8.4. Abstracting from ancillary details, `spawn` internally works as follows:

1. combines new process keys with previous ones (from the device itself) and those from direct neighbours,

2. maps the resulting keyset by running `process` in an aligned way w.r.t. the process keys; and finally
3. filters results upon the boolean status value.

Crucially, filtering results prevents writing exports (this is achieved through the `vm` calls): so, filtered processes are not broadcast to neighbours—this mechanism ultimately impacts the spatiotemporal evolution of a process.

8.4 Programming with Aggregate Processes: Techniques and Patterns

In the following, we discuss programming and management of aggregate processes activated through `spawn`. We start from the basics (process definition, lifecycle and boundary management) and then introduce more complex examples in order to delineate the principle behind an "aggregate process API", prepare for the case studies that follow—concretely showing how composition of collective behaviour could support the engineering of pervasive applications.

8.4.1 Process definition

Defining a (type of) process merely consists of defining a function that can be passed as the `process` parameter to `spawn`. It must be a curried function from a process key `K`, an argument `A`, to a tuple result `(R,Boolean)`—for some choice of `K`, `A`, `R` made statically at a particular call of `spawn`.

It is good practice to define custom types for `K`, `A`, and `R`:

```
case class PID(id: Int)(val initiator: ID)
case class PArgument(arg: Int)
case class PResult(result: String)
```

Therefore, a process definition could take the following schema:

```
// Method syntax
def myProcessLogic(pid: PID)(parg: PArgument): (PResult,Boolean) = {
  val result: PResult = // compute result
  val stay: Boolean = // compute logic for process boundary/lifecycle
  (result, stay)
}

// Function syntax
val myProcess = (pid: PID) => (parg: PArgument) => { /* ... */ }
// or, from a method:
val myProcess: PID => PArgument => PResult = myProcessLogic _
```

Once we have defined a process function, we can use `spawn` to create process instances:

```
spawn[PID,PArgument,PResult](myProcess, ...)
```

8.4.2 Process generation (lifecycle management 1/2)

Generating process instances is just a matter of creating a field of keysets that become non-empty as soon as some "triggering" space-time event has been recognised. Examples include spatial conditions on sensors data and computation, timers firing, and so on [Vir+18].

Consider the following, simple process definition.

```
type K = Int // The type (alias) of process keys
type A = Int // The type (alias) of process arguments
type R = (Int, Boolean) // The type (alias) of process return

def m(k: K)(a: A): R = (k + a, true) // true means: always participate
val p = m _
```

A trivial example could leverage a constant, uniform field with full domain.

```
val keySet = Set(1)
val argument = 2
val processes = spawn(p, keySet, argument)
```

In this case, a single process instance gets activated everywhere, and repeatedly applied (on a round by round basis) against a constant argument: for every device (everywhere), `processes` is always (everytime) a `Map(1->3)`. Of course, we can

spawn multiple instances of the same process type in a single `spawn`, and provide a non-uniform argument field. For instance, expression

```
// Remember: mid is the field of local device IDs
spawn(p, newKeys = Set(1,2), args = mid)
```

yields a constant field that is locally `Map(1->1+`δ`,2->2+`δ`)` for any device δ.

Things get more interesting when the keyset field is non-uniform. Consider a connected system of three devices δ_1, δ_2, δ_3. Since process keys are automatically propagated to neighbour devices (in a hop-by-hop fashion), expression

```
spawn(p, newKeys = mid, args = 0)
```

will stabilise to a field that is everywhere `Map(`δ_1`->`δ_1`,`δ_2`->`δ_2`,`δ_3`->`δ_3`)`. In this case, the"source" or "generator" of process with PID δ_i is the device δ_i itself. The time it takes for a process to spread depends on the timing of round execution and communication acts in the different devices. Now, suppose the system gets split into two partitions (δ_1, δ_2) and (δ_3), and that, later, the latter is joined by a device δ_4: under these circumstances, the output will remain the same for δ_1, δ_2 whereas δ_3 and δ_4 will both output `Map(`δ_1`->`δ_1`,`δ_2`->`δ_2`,`δ_3`->`δ_3`,`δ_4`->`δ_4`)`.

Typically, processes are generated by specific devices, when specific conditions come true. This is modelled by a keyset field which is empty everywhere in space-time except in locations where the event is recognised A schema is the following,

```
val event: Boolean        = // ...
val generateKey: Any => K = // ...
val keys: Set[K] = if(event){ Set(generateKey(???)) } else { Set.empty }

spawn(???, keys, ???)
```

where you generally need a way to generate a process key to uniquely identify a process instance with the particular occurrence of the event.

As mentioned before, process keys work both as process identities and as construction parameters. Consider the following process modelling a gradient computation.

```
def gradientProcess(source: ID)(obstacle: Boolean): Double =
  branch(!obstacle){ gradient(source==mid) }{ Double.PositiveInfinity }
```

In this case, since the `ID` of the source is used to identify a process instance, you cannot have more than one gradient process per source. Now, suppose you want to preserve the same semantics but also keep track of the device who generated the process (which is not necessarily the source of the gradient): you do not want process identity to depend on the generator, so your key data type must carry the additional information while handling identity (i.e., equality) like in the previous example. For this purpose, the following Scala `case class` idiom comes handy.

```
case class PID(source: ID)(val generator: ID)
```

Finally, a clarification is needed, regarding the semantics of `spawn` and the peculiar execution model of round-by-round field computations (as they especially relate to branching). Construct `spawn` differs from traditional "thread spawning" constructs like Erlang's `spawn` or Java's `Thread.start()`, in that a ScaFi's `spawn` expression needs to always be evaluated in each round in order to carry through active process instances. That is, in the following program,

```
branch(someCondition){
  // ...
  spawn(???, ???, ???)
}{
  // ...
  spawn(???, ???, ???)
}
```

taking a branch will cause the destruction of all the process instances `spawn`ed in the other branch.

Time tracking in ScaFi Basic techniques for process generation include space-time event recognition and time-wise scheduling. For the purpose, building block `T` is used to model the passing of time in field computations [Vir+18]. In ScaFi, it can be implemented as follows.

```
def T(init: Double, floor: Double, decay: Double => Double): Double =
  rep(init) { v => Math.min(init, Math.max(floor, decay(v))) }

def T(init: Double, delta: Double): Double =
  T(init, 0.0, (t: Double) => t - delta))
```

Operator `T` works by keeping track of the remaining time (starting from `init`) via

construct `rep`, and then using the provided function `decay` to enact the passing of time until `floor` is hit. A derived version based on a `delta` step can be straightforwardly defined. Built-in, local sensor `dt()` is used to locally keep track of time passed since the previous computation round. Using `T`, it is trivial to spawn a process once after some delay:

```
val newPids = mux(T(100, dt())==0){/* gen new keyset */}{ Set() }
```

The key thing to understand is that such a "once timer" restarts any time the corresponding computation is "re-entered"; or, in other words, it is refreshed when the corresponding computation is not executed (since its `rep` node, by disappearing from the vtree, loses its memory); hence, a clock based on a cyclic timer can be implemented as follows:

```
def clock(len: Long, decay: Long): Long =
  rep((0L,len)){ case (k,left) => // Function defined by pattern matching
    branch (left == 0){ (k+1,len) }{ (k, T(len, decay).toLong) }
  }._1 // "_1" projects to the first element of the tuple
```

Such a clock can be used for periodically spawning processes: see, e.g., the `replicated` example below.

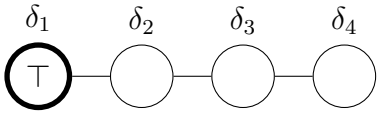
8.4.3 Process expansion/shrinking (boundary management)

Notice that a condition for process generation is that the generating device does not immediately quit itself. By `spawn`, every process instance is *automatically propagated by all the participating devices to their neighbours*. Such a propagation does not occur only if the device returns status `false` (which means that the device does not want to participate in that process instance). Therefore, it is possible to regulate the shape of such "computational bubble" by dictating conditions by which a device must return status `false` (i.e., meaning *external* to the bubble)—as mentioned, this indicates the willingness to *stop* computing (i.e., participate in) the process. That is, only devices that return status `true` (i.e., *internal*) will propagate the process.

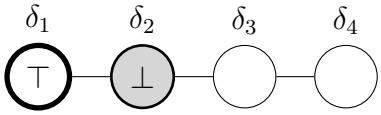
Moreover, such a propagation happens continuously: so, a device that exited

from a process may execute it again in the future (if its neighbours are still internal to that process). In particular, the *border* (or *fence*) of a process bubble is given by the set of all the devices that are external but have at least one neighbour which is internal. As long as a node is in the fence, it continuously re-acquires and immediately quits from the process instance: this *continuous evaluation of the border* is what ultimately enables a spatial extension of the process bubble (*expansion*). Conversely, a process bubble gets restricted (*shrinking*) when internal nodes become external.

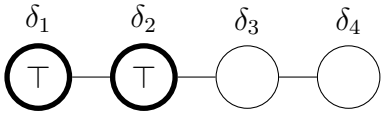
As an example of expansion and shrinking, consider the following system evolution, where a process instance is generated at δ_1 and $\top$ (resp. $\bot$) represents `true` (resp. `false`) status.



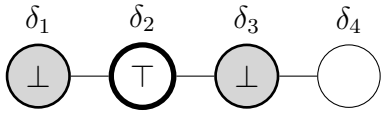
Generation at δ_1 (process will spread to δ_2)



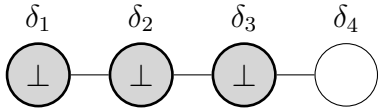
Border formation: δ_2 forms the external fence



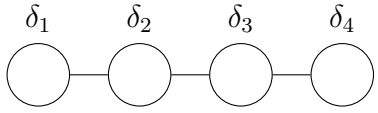
Process expansion: δ_2 decides to join and becomes internal (process will spread to δ_3)



Process shrinking: δ_1 leaves (while δ_3 enters the external border to monitor expansion)



Pre-termination: all the devices are external, so the process bubble can vanish



The process disappears (but might be regenerated in future)

8.4.4 Process termination (lifecycle management 2/2)

As we have seen, a process instance *terminates* when all the devices quit by returning status `false`. Implementing process termination may not be trivial,

since proper (local or global) conditions must be defined so that the "collapsing force" can overtake the "propagation force"; i.e., precautions should be taken so that external devices do not re-acquire the process: the border should steadily shrink, also considering temporary network partitions and transient recoverable failures from devices. In the following pages, we will develop a higher-level support to process termination based on "termination signals".

Example: spatiotemporally limited processes It is often useful to run processes on a limited subset of the devices (e.g., those contained within a certain range from the process generator), for a limited amount of time. In order to support this, a process should carry information about the generation location, the distance from the generation location, the time of generation, and the time that has elapsed since generation. Border and lifecycle management should manage and predicate on such information.

```
case class PID(pid: String)
              (val generator: ID, val startTime,
               val lifetime: Long, val maxRange: Double)

type K = PID
type A = Unit // we are not interested in any runtime argument
type R = Int

def logic(k: K)(a: A): R = 0 // trivial
def lifecycle(k: K)(a: A): Boolean =
  time()-k.startTime < k.lifetime &
    gradient(k.generator==mid) <= k.maxRange

// This utility function merges logic with lifecycle functions into one
def combine[K,A,R1,R2](f1:K=>A=>R1)(f2:K=>A=>R2):K=>A=>(R1,R2) =
  k => a => (f1(k)(a), f2(k)(a))

spawn[K,A,R](combine(logic)(lifecycle), /* newKeys */, ())
```

In the above example, devices call themselves out when the process time exceeds `lifetime` or the distance computed by the gradient exceeds `maxRange`.

Notice that, in circumstances like this, the logic of computation can be completely separated from process border and lifecycle management; in these cases, program design can benefit from *separation of concerns*, adopting a *single-responsibility principle* while functional composition enables creation of a full process definition for `spawn`. Moreover, with careful design, this enables *reusability* of

lifecycle strategies:

```
trait STLimitedProcessKey {
  val generator: ID
  val startTime: Long
  val lifetime: Long
  val maxRange: Double
}

def STLifecycle[K,A,R](k: K <: STLimitedProcessKey)(a: A): Boolean =
  time()-k.startTime < k.lifetime &
    gradient(k.generator==mid) <= k.maxRange
```

The above `STLifecycle` can be combined with *any* process logic over process keys that conform to `STLimitedProcessKey`[1].

8.4.5 Process abstraction

Using functional abstraction, it is possible to define high-level behaviours that provide a clean interface hiding the complexity of internal process management.

Example: time replication In [PBV16], a technique based on time replication for improving the dynamics of gossip is presented. It works by keeping k running replicates of a gossip computation executing concurrently, each alive for a certain amount of time. New instances are activated with interval p, staggered in time. The whole computation always returns the result of the oldest active replicate. This is intended to improve the dynamics of algorithms, providing an intrinsic refresh mechanism that smoothly propagates to the output. With `spawn`, it is trivial to design a `replicated` function that provides process replication in time.

```
def replicated[A,R](proc: A => R)(argument: A, p: Double, k: Int) = {
  val lastPid = clock(p, dt())
  spawn[Long,A,R](pid => arg => (proc(arg), pid > lastPid+k),
                  Set(lastPid), argument)
} // returns a Map[Long,R] from replicate IDs to corresponding values
```

`clock` is a distributed time-aware counter [PBV16] (whose synchronicity depends on the implementation) yielding an increasing number i at each interval p that represents the PID of the i-th replica. Notably, in this case, every device can locally

[1]Notice that Scala provides mechanisms, such as *structural types* or the *pimp-my-library pattern* [OMO10], to avoid the requirement of explicit, apriori trait implementation.

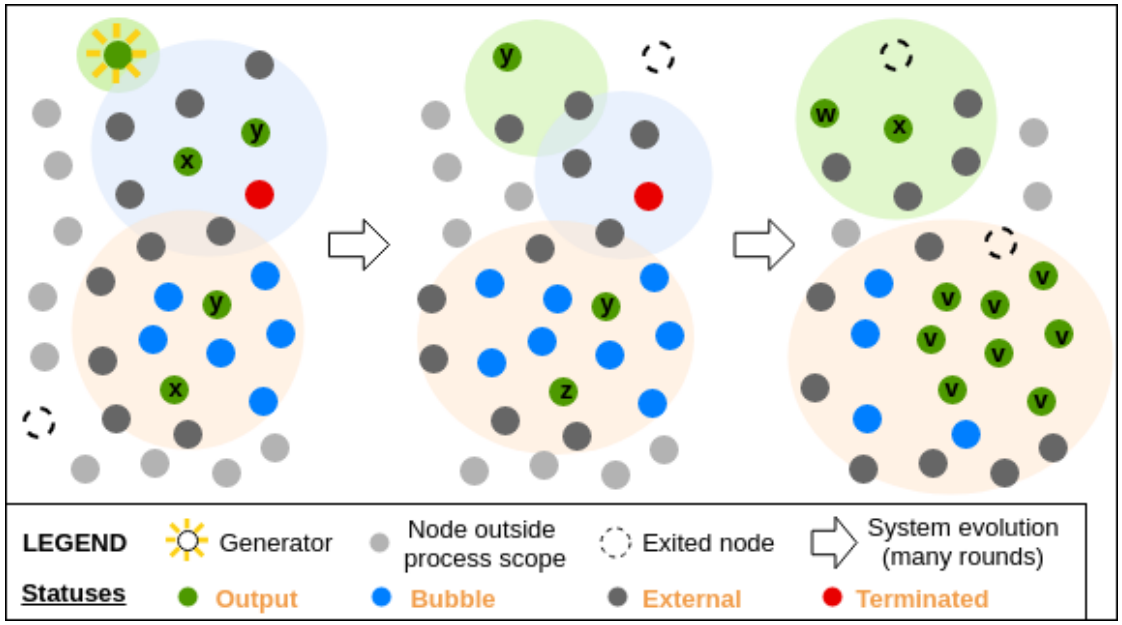


Figure 8.5: Graphical example of the evolution of a system of processes and the role of statuses in `statusSpawn`. The green bubble springs into existence; the blue bubble dissolves after termination is initiated by a node; the orange bubble expands. Only output nodes will yield a value. Bubbles may of course overlap (i.e., a node may participate, with different statuses, to multiple processes) and the dynamics can be arbitrarily complex (because of mobility, failures, and local decisions).

determine when it must quit a process instance; moreover, the exit condition based on PID numbering (`pid > lastPid+k`) prevents process reentrance. Section 8.5.2 provides an empirical evaluation of the behaviour of function `replicated`.

8.4.6 Process interaction

The most basic means to make aggregate processes interact is by *piping* the output of a process into the input of another.

```
val p1s = spawn[K1,A1,R1](???, ???, ???) // output is a Map[K,R]

val someDefault: R1 = ???
val arg = p1s.headOption.getOrElse(someDefault)
type A2 = R1
val p2s = spawn[K2,A2,R2](???, ???, arg)
```

Moreover, the following programming idiom can be used in the case of mutually feeding processes, `spawn`s in different scopes, or when a "program-wide" communication structure is desired:

```
case class Msg[V,From,To](body: V, from: From, to: To)
type MBox = List[Any]
type PostOffice = Map[Any,MBox]

rep[PostOffice](Map.empty)(msgs => {
  // ...
  spawn(???, ???, Args1(???, msgs))
  // ...
  spawn(???, ???, Args2(???, msgs))
  // ...
  msgs
})
```

8.4.7 More expressive process definitions

Now, we show how to support more declarative process definitions by leveraging expressive "statuses". First, we define the concrete `Status`es:

```
trait Status

case object ExternalStatus extends Status   // External to the bubble
case object BubbleStatus extends Status     // Within the bubble
case object OutputStatus extends Status     // Within the bubble + output
case object TerminatedStatus extends Status // Willingness to shutdown

val External: Status = ExternalStatus
val Bubble: Status = BubbleStatus
val Output: Status = OutputStatus
val Terminated: Status = TerminatedStatus
```

Then, we capture a process output not as a tuple `(T,Boolean)` but as a tuple `(T,Status)`, which we render as an algebraic data type to provide useful implicit conversions to the former form (leveraging the power of Scala).

```
case class POut[T](result: T, status: Status)
object POut {
  // Implicit definition to map POut to (T,Boolean)
  implicit def toBasicSpawnTuple[T](pout: POut[T]): (T,Boolean) =
    (pout.result, pout.status!=External)
  // Conversion between process computation definitions
  implicit def fto[K,A,R](proc: K => A => POut[R]): K=>A=>(R,Boolean) =
    k => a => toBasicSpawnTuple(proc(k)(a))
}
```

At this point, we can handle termination by mapping process computation definitions; we can employ a simple shutdown algorithm that distributes the termination signal to neighbours, closes the process in a device (by going `External`) when all the neighbours have received such signal, and prevents re-acquisition of the process if any neighbour presents the termination signal.

```
def handleTermination[T](out: POut[T]): POut[T] = {
  rep[(Boolean,Int,POut[T])]((false,0,out)){
    (terminated,k,res) =>
      val mustTerminate = out.status==Terminated |
                          includingSelf.anyHood(nbr{terminated})
      val mustExit = includingSelf.everyHood(nbr{mustTerminate})
      (mustTerminate, // true if observed termination signal
      1, // flag (k=0 only in the first round for this process)
      if(mustExit || (mustTerminate && k==0))
        (out.result, External) // enforce quit
      else
        out // just pass given (output,status) through
      )
  }._3
}
```

Output filtering is achieved by mapping results to optional values that are present only when the device has status `Output`; however, this also requires a filtering outside the call to `spawn`. Function

```
def handleOutput[T](out: POut[T]): POut[Option[T]] = out match {
  case POut(res, Output) => POut(Some(res), Output)
  case POut(_, s) => POut(None, s)
}
```

maps process results (of type `T`) to `Option[T]` values, present (`Some` constructor) or not (`None` constructor) based on whether status is `Output` or not, resp.

Finally, a higher-level "spawn" function `statusSpawn` can be defined as follows:

```
def statusSpawn[K, A, R](process: K => A => POut[R],
                         newKeys: Set[K],
                         args: A): Map[K,R] =
  spawn[K,A,Option[R]](
    k => a => handleOutput(handleTermination(process(k)(a))),
    params,
    args).collectValues { case Some(p) => p }
```

where `handleOutput` and `handleTermination` wrap the given `process` (which must yield a `POut[T]` value), and only `Option[R]` values that are present are kept.

Example: limited multi-gradient The problem described in Section 8.3.1 of activating a spatially-limited gradient computation for each device where sensor `isSrc` gives true, and deactivating it when it stops doing so, can be solved as follows:

```
def multiGradient(isSrc: Boolean, maxExtension: Double) =
  statusSpawn[ID,Double,Double](src => limit =>
    gradient(src==mid,nbrRange) match { // consider the usual gradient
      case g if src==mid && !isSrc => (g, Terminated) // close on unsource
      case g if g>limit => (g, External)              // out of bubble
      case g => (g, Output)                           // in bubble + get
    },
    newKeys = if(isSrc) Set(mid) else Set.empty,
    args = maxExtension
  )
```

where we also show the "closure idiom", by which a process behaviour is defined as a closure, i.e., a function closing over its environment (in this case, parameter `isSrc`).

8.5 Evaluation

8.5.1 Case study: opportunistic messaging

Motivation The possibility of communicating by delivering messages regardless the presence of a conventional Internet access has recently gained attention as a mean to work around censorship (`http://archive.is/C3ni0`) as well as in situations with limited access to the global network—e.g., in rural areas, or during

urban events when the network capability is overtaken. We here consider a simple messaging application where a source device (aka *sender*) wants to deliver a payload to a peer device (aka *recipient*, *target*, or *destination*) in a hop-by-hop fashion by exploiting nearby devices as relays. The source device only knows the identifier of its recipient: it is not aware of its physical location, nor of viable routes. Our goal is to show how aggregate processes can support this kind of application (with multiple concurrent messages) while limiting the number of devices involved in message delivery, leading to bandwidth savings (and energy savings in turn).

Opportunistic chat implementation The idea of the case study is to activate an aggregate process for each message sent from a source node to a destination node, and to limit the extension to such process instance to a small subset of the devices belonging to the whole system. A simple algorithm to do so involves creating information flows from the source node to a "central" node, and from the central node to the recipient node. Once the recipient has received the message, the message delivery process must be closed.

An implementation can be as follows. First, we model the data, i.e., the message (which also represents the PID), and coordination data used for directing the shape of the process (which also represent the runtime arguments):

```
case class Msg(src: ID, dest: ID, str: String)
case class ChatArgs(parentToCentre: ID, dependentNodes: Set[ID])
```

where `parentToCentre` is, for each device, the identifier of the closest neighbour to the central node, and `dependentNodes` are the neighbours which consider the current device as "parent". Then, we define the process computation logic:

```
def chatProcessLogic(msg: Msg)(args: ChatArgs): POut[Msg] = {
  // Boolean field expressing a path from the source of a message
  // to a centre node
  val srcToCentrePath = msg.src==mid | includingSelf.anyHood {
    nbr(args.parentToCentre) == mid
  }
  val destToCentrePath = args.dependentNodes.has(msg.target)
  val inRegion = srcToCentrePath || destToCentrePath
  POut(result=msg, status=branch(mid == msg.target){
    justOnce(Output, thereafter=Terminated)
  }{ mux(inRegion){ Bubble }{ External } })
}
```

where the target of the message (`msg.target`) has status `Output` at first and then `Terminated`, and only nodes for which field `inRegion` is locally true have status `Bubble` and hence participate in the message delivery process for `msg`. Finally, we define a `chat` function that leverages `statusSpawn`:

```
def chat(centre: ID, newMsgs: Set[Msg]): Iterable[Msg] = {
  val (_, parentToCentre) = gradientWithParent(centre == mid)
  val dependentNodes = rep(Set.empty[ID]){ case (s: Set[ID]) =>
    excludingSelf.unionHoodSet[ID](
      mux( nbr{parentToCentre}==mid ){ nbr(s) }{ Set.empty[ID] }
    ) + mid } // nodes whose path towards centre passes through me

  statusSpawn[Msg,ChatArgs,Msg](
    process = chatProcessLogic(_), // note: m(_) turns method m to lambda
    newKeys = newMsgs,
    args    = ChatArgs(parentToCentre, dependentNodes)).values
}
```

where the device used as centre and new messages to be sent are externally provided through parameters `centre` and `newMsgs`, respectively. The output of function `chat` is the field of the collections of messages that have been currently received at the recipient devices.

Experimental setup We compare two aggregate implementations of such messaging system. The first implementation, called *flood chat*, simply broadcasts the payload to all neighbours. In spite of an in-place garbage collection system, however, this strategy may end up dispatching the message towards directions far-off the optimal path, burdening the network. The second implementation, *spawn chat*, leverages `spawn` in order to reduce the impact on the network infrastructure by

electing a node as coordinator, then creating an aggregate process connecting the source and the coordinator and the coordinator and the destination, and finally delivering the message along such support. In this experiment, we naively choose a coordinator randomly, but better strategies could be deployed to improve over this configuration. The experiment is simulated on a mesh network of one thousand devices randomly deployed in the urban area of Cesena, in Italy. We simulate the creation and delivery of messages among randomly chosen nodes, with one message per second generated on average by the whole network in time window $[0, 250]$; devices execute rounds asynchronously at an average of 1Hz. In each experiment, we generate a different random displacement, different message sources and destinations, and different random seeds for the drift distributions. We gather a measure of QoS and a measure of resource usage. We use the probability of delivering a message with time as a QoS measure, and we measure the number of payloads sent by each node as a measure of impact on performance. In doing so, we suppose payload makes up for the largest part of the communication (as is typically the case when multimedia data are exchanged).

Results Figure 8.6 shows experimental results. The two implementations achieve a very similar QoS, with the flood implementation being faster on average. This is expected, as flooding the whole network also implies sending through the fastest path. The difference, however, is relatively small and, on the contrary, we see the *spawn chat* affords a dramatic decrease in bandwidth usage (by properly constraining the expansion of message delivery bubbles), despite the simplistic selection of the coordinating device.

8.5.2 Case study: drone swarm reconnaissance

Motivation Performing reconnaissance of areas with hindrances to access and movement such as forests, steep climbs, or dangerous conditions (e.g. extreme weather and fire) can be a very difficult task for ground-based teams. In those cases, swarms of unmanned airborne vehicles (UAVs) could be deployed to quickly gather information [Bea+18]. One scenario in which such systems are particularly interesting is fire monitoring [Cas+06]. With this case study, we show how aggregate processes enable easy programming of a form of gossip that supports a precise

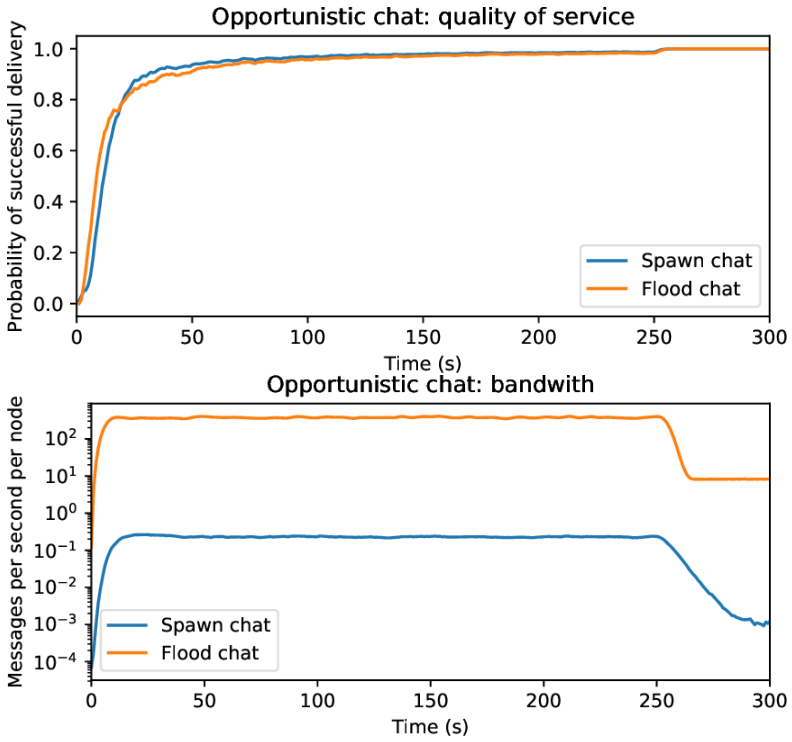


Figure 8.6: Evaluation of the opportunistic chat algorithms. The figure on top shows similar performance for the two algorithms, with the *flood chat* featuring a slightly better delivery time for the payloads (as it intercepts the optimal path among others). However, as the bottom figure depicts, *spawn chat* requires orders of magnitude less resources due to the algorithm executing on a bounded area (i.e., by involving only a subset of system devices for each message delivery process).

collective estimation of risk in dynamic scenarios.

Experimental setup In this case study, we simulate a swarm of 200 UAVs in charge of monitoring the area of Mount Vesuvius in Italy, which has been heavily hit by wildfires in 2017 (`http://archive.is/j3lsm`). Multiple simulation runs are performed for different random seeds. UAVs follow a simple exploration strategy: they all start from the same base station on the southern side of the volcano, they visit a randomly generated sequence of ten waypoints, and once done they come back to the station for refuelling and maintenance. UAVs sense their surroundings once per second and assess the local situation by measuring the risk of fire. The goal of the swarm is to agree on the area with the highest risk of fire and report the information back to the station for deployment of ground inter-

```
def gossipNaive[T](value: T)(implicit ev: Bounded[T]) = rep(value)(max =>
    ev.max(value, maxHoodPlus(nbr(ev.max(max, value)))))

def gossipGC[T](value: T)(implicit ev: Bounded[T]) = {
  val leader = S(grain = Double.PositiveInfinity, nbrRange)
  valueBroadcast(leader, C[Double,T](
    potential = gradient(leader),
    acc = ev.max(_,_), local = value, Null = ev.bottom))
}

def gossipReplicated[T:Bounded](value: T, p: Double, k: Int) =
  (replicated{ gossipNaive[T] }(value,p,k) // returns a Map[Long,Double]
   + (Long.MaxValue -> value) // default, lowest-priority entry of the map
  ).minBy[Long](_._1)._2 // projects the value of instance with min PID
```

Figure 8.7: Code of the gossip algorithms used in the reconnaissance case study.

vention. A snapshot of the drones performing the reconnaissance is provided in Figure 8.8. We are not concerned with realistic modelling of fire dynamics: we designed the risk of fire to be maximum in a random point of the surveyed area for 20 minutes; the risk then drops (e.g. due to a successful fire-fighting operation), with the new maximum (lower than the previous) being in another randomly generated coordinate; after further 20 minutes the risk sharply increases again to on a third random coordinate. We compare three approaches: *(i) naive gossip*, a simple implementation of a gossip protocol; *(ii) S+C+G*, a more elaborated algorithm – based on self-stabilising building blocks [Vir+18] – that elects a leader, aggregates the information towards it, then spreads it to the whole network by broadcast; *(iii) replicated gossip*, which replicates the first algorithm over time (as per [PBV16]) and whose implementation, shown in Figure 8.7, uses function `replicated` (defined in Section 8.4.5 upon `spawn`).

Results Results are shown in Figure 8.9. The naive gossip algorithm quickly converges to the correct value, but then fails at detecting the conclusion of the dangerous situation: it is bound to the highest peak detected, and so it is unsuitable for evolving scenarios. S+C+G can adapt to changes, but it is very sensitive to changes in the network structure: data gets aggregated along a spanning tree generated from the dynamically chosen coordinator, but in a network of fast-moving airborne drones such structure gets continuously disrupted. Here the `spawn`-based

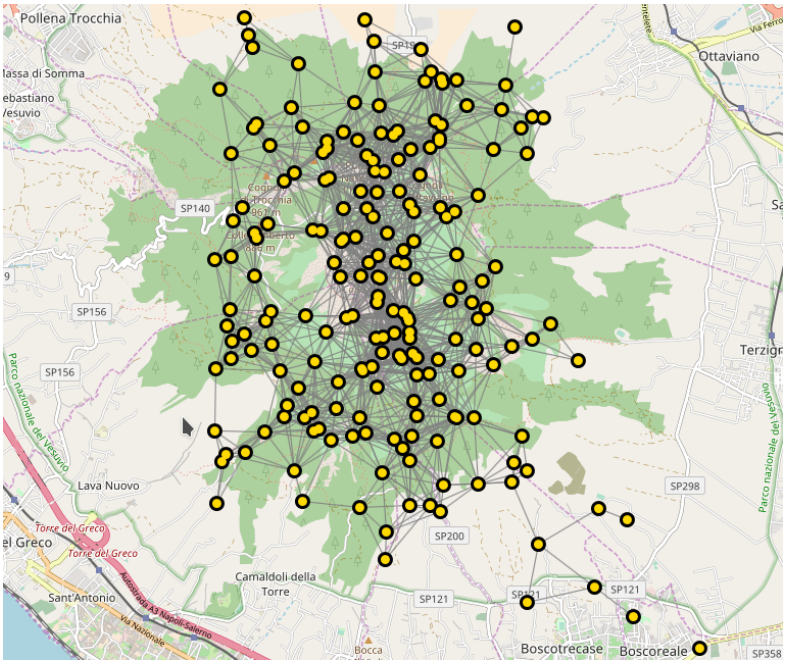


Figure 8.8: Snapshot of the UAV swarm surveying the Vesuvius area as simulated in Alchemist. Yellow dots are UAVs. Grey lines depict direct drone-to-drone communication. Drones travel at an average speed of $130\frac{km}{h}$, in line with the cruise speed performance of existing military-grade UAVs (see `http://archive.is/8zar5`), and communicate with other drones within 1km distance in line-of-sight. Forming a dynamic mesh network using UAV-to-UAV communication is feasible [FB08], although challenging [GJV16].

replicated gossip performs best, as it conjugates the stability of the naive gossip algorithm with the ability to cope with reductions in the sensed values. The algorithm in this case provides underestimates, as it reports the highest peak sensed in the time span of validity of a replicate, and drones rarely explore the exact spot where the problem is located, but rather get in its proximity.

8.6 Final Remarks

In this chapter, a notion of aggregate processes is proposed and implemented in order to model dynamic, concurrent collective adaptive behaviours carried out by dynamic formations of devices—hence extending over field calculus and ScaFi.

This work draws inspiration from previous work in the context of the Pro-

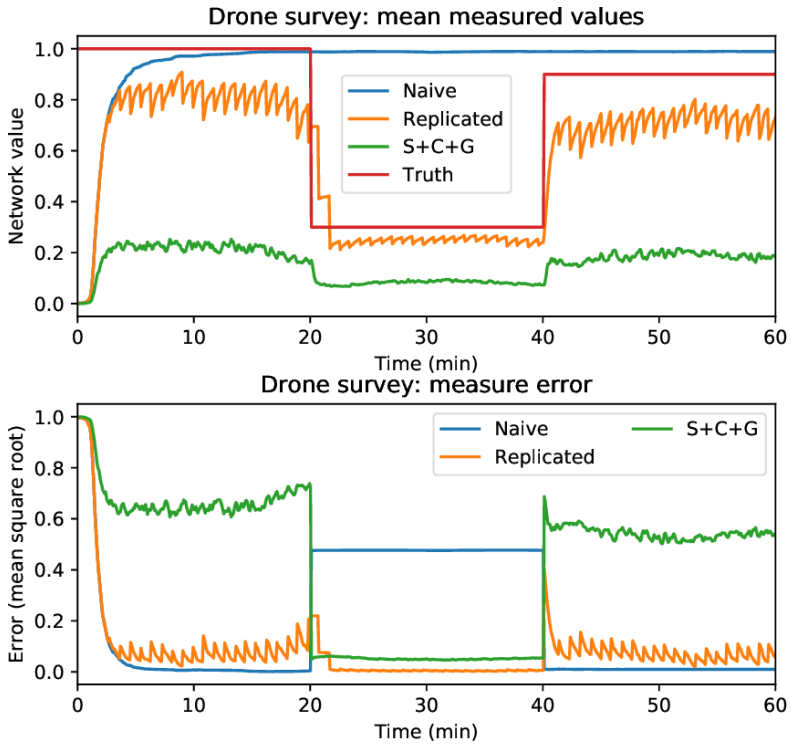


Figure 8.9: Evaluation of the gossip algorithms in the UAV reconnaissance scenario. The figure on top shows expected values and measures performed by the competing algorithms. The bottom figure measures the error as root mean square: $\sqrt{\frac{\sum_n (v_n - a)^2}{n}}$ where n device count, a actual value, and v_n value at the n-th device. The naive gossip cannot cope with danger reduction, S+C+G cannot cope with the volatility of the network, while replicated gossip provides a good estimate while being to cope with changes.

telis aggregate programming language [PVB15], especially [PBV16], where a gossip algorithm based on overlapping replicates is proposed, and [Fra+17], where a `multiInstance` pattern is described—both implemented in terms of an `alignedMap` building block. Construct `alignedMap`, though not formalised, appears similar to the `spawn` construct: it has signature `alignedMap(keys,filter,f, default)` and apparently works by running `f` in an aligned way on the provided `keys` (filtered by `filter`), returning `default` when no value is available. Construct `spawn` differs from `alignedMap` in that, the former (i) has been given precise semantics, as covered in this chapter, (ii) is typed, (iii) locally keeps the state of active keys from round to round, and (iv) provides automatic propagation of purely local keys

to neighbours, hence simplifying border management.

The aggregate process abstraction is related to works on distributed coordination (Chapter 3), spatial computing, and collective adaptive systems (Chapter 4), as well as to organisational paradigms in multi-agent systems [HL04]. Indeed, with aggregate processes, it is possible to program the logic of group formation to implement various grouping strategies. In the messaging case study, e.g., a dynamic, goal-directed *team* of devices is formed just to connect senders with recipients, dissolving when the task is completed. Work on attributed-based communication [De +14] is also related: these approaches leverage information chunks (attributes) to dynamically bind components into ensembles. Though different in abstraction, the `spawn` construct can be used to replicate the same behaviour for group membership, where an ensemble is captured by a process instance and process keys are sets of attributes.

References

[Aud+16] Giorgio Audrito, Ferruccio Damiani, Mirko Viroli, and Roberto Casadei. "Run-Time Management of Computation Domains in Field Calculus". In: *Foundations and Applications of Self* Systems, IEEE International Workshops on*. IEEE. 2016, pp. 192–197.

[Aud+17] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. "Compositional Blocks for Optimal Self-Healing Gradients". In: *Self-Adaptive and Self-Organising Systems (SASO), IEEE International Conference on*. IEEE. 2017.

[Aud+18] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. "Space-time universality of field calculus". In: *International Conference on Coordination Languages and Models*. Springer. 2018, pp. 1–20.

[Aud+19] Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. "A Higher-Order Calculus of Computational Fields". In: *ACM Transactions on Computational Logic* 20.1 (2019), pp. 1–55. DOI: 10.1145/3285956.

[Bea+18] Jacob Beal, Kyle Usbeck, Joseph Loyall, Mason Rowe, and James Metzler. "Adaptive Opportunistic Airborne Sensor Sharing". In: *ACM Trans. Auton. Adapt. Syst.* 13.1 (Apr. 2018), 6:1–6:29. ISSN: 1556-4665. DOI: 10.1145/3179994. URL: http://doi.acm.org/10.1145/3179994.

[Cas+06] David W. Casbeer, Derek B. Kingston, Randal W. Beard, and Timothy W. McLain. "Cooperative forest fire surveillance using a team of small unmanned air vehicles". In: *International Journal of Systems Science* 37.6 (2006), pp. 351–360. DOI: 10.1080/00207720500438480. URL: https://doi.org/10.1080/00207720500438480.

[De +14] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. "A Formal Approach to Autonomic Systems Programming: The SCEL Language". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 9.2 (2014), 7:1–7:29. DOI: 10.1145/2619998.

[DV15] Ferruccio Damiani and Mirko Viroli. "Type-based Self-stabilisation for Computational Fields". In: *Logical Methods in Computer Science* 11.4 (2015).

[FB08] E.W. Frew and T.X. Brown. "Airborne Communication Networks for Small Unmanned Aircraft Systems". In: *Proceedings of the IEEE* 96.12 (2008), pp. 2008–2027. DOI: 10.1109/jproc.2008.2006127. URL: https://doi.org/10.1109/jproc.2008.2006127.

[Fra+17] Matteo Francia, Danilo Pianini, Jacob Beal, and Mirko Viroli. "Towards a Foundational API for Resilient Distributed Systems Design". In: *International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. IEEE, 2017. DOI: 10.1109/fas-w.2017.116.

[GJV16] Lav Gupta, Raj Jain, and Gabor Vaszkun. "Survey of Important Issues in UAV Communication Networks". In: *IEEE Communications Surveys & Tutorials* 18.2 (2016), pp. 1123–1152. DOI: 10.1109/comst.2015.2495297. URL: https://doi.org/10.1109/comst.2015.2495297.

[HL04] Bryan Horling and Victor Lesser. "A survey of multi-agent organizational paradigms". In: *The Knowledge engineering review* 19.4 (2004), pp. 281–316.

[LLM17] Alberto Lluch-Lafuente, Michele Loreti, and Ugo Montanari. "Asynchronous Distributed Execution Of Fixpoint-Based Computational Fields". In: *Logical Methods in Computer Science* 13.1 (2017). DOI: 10.23638/LMCS-13(1:13)2017. URL: https://doi.org/10.23638/LMCS-13(1:13)2017.

[OMO10] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. "Type classes as objects and implicits". In: vol. 45. 10. Association for Computing Machinery (ACM), Oct. 2010, p. 341. DOI: 10.1145/1932682.1869489. URL: https://doi.org/10.1145/1932682.1869489.

[PBV16] Danilo Pianini, Jacob Beal, and Mirko Viroli. "Improving Gossip Dynamics Through Overlapping Replicates". In: *Proceedings of the 18th International Conference on Coordination Models and Languages*. Vol. 9686. Lecture Notes in Computer Science. Springer, 2016, pp. 192–207. DOI: 10.1007/978-3-319-39519-7_12.

[PVB15] Danilo Pianini, Mirko Viroli, and Jacob Beal. "Protelis: practical aggregate programming". In: *Symposium on Applied Computing*. ACM. 2015, pp. 1846–1853. DOI: 10.1145/2695664.2695913.

[Vir+18] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. "Engineering Resilient Collective Adaptive Systems by Self-Stabilisation". In: *ACM Transaction on Modelling and Computer Simulation* 28.2 (2018), 16:1–16:28. ISSN: 1049-3301. DOI: 10.1145/3177774.

Chapter 9

Aggregate Computing Platforms

Design is to design a design to produce a design.

John Heskett

Contents

Building distributed systems is difficult, for many challenges beyond plain connectivity need to be addressed (as briefly covered in Chapter 3). Therefore, for non-trivial applications, various layers of software need to be put in place in order to fill the *abstraction gap*, i.e., the conceptual and technical distance between application-level abstractions and the underlying platform (hardware plus infrastructural software—see Chapter 6). Moreover, this part should not dramatically change for similar applications, hence making the case for *reusability*. This reusable piece of software is generally called a *middleware* [SS02], since it sits in the middle

between lower layers and applications. However, different applications or different hardware platforms might require different middleware software: the tension between generality and specificity is a critical concern in middleware design [HM06].

Aggregate computing, as a programming paradigm, could be used to build applications. However, the ability to express and interpret aggregate programs (as covered in Chapters 7 and 8) – e.g., as provided by the ScaFi DSL – is not per se sufficient: in order to build a distributed "aggregate system", other concerns need to be dealt with, ranging from communication, security, and of course the management of aggregate dynamics. In other words, aggregate computing-based applications should be supported by an aggregate computing middleware. Accordingly, this chapter provides an analysis of the problem and presents a proof-of-concept implementation included in the ScaFi toolkit.

9.1 Analysis of Aggregate Computing Platforms

9.1.1 Preliminary definitions: main entities and artefacts

In order to foster a systematic characterisation of the problem domain, and to be more precise in the prose, we introduce the following terms.

- An *aggregate contract* defines the behaviour of the participants of an aggregate.
- An *aggregate program* is a executable piece of *aggregate logic* implementing an *aggregate contract*, and is expressed, e.g., through a field programming language (like Protelis or ScaFi).
- An *aggregate application* is an aggregate program plus configuration and other software that altogether provide some functionality to users.
- The *aggregate computing middleware* (or *platform*) is a piece of software that supports the development, deployment, and execution of aggregate applications; i.e., the middleware fills the abstraction gap between aggregate applications and the underlying platform or operating system, and it also hides heterogeneity, allowing applications to be described independently of the specific hardware, communication technology, and operating systems.

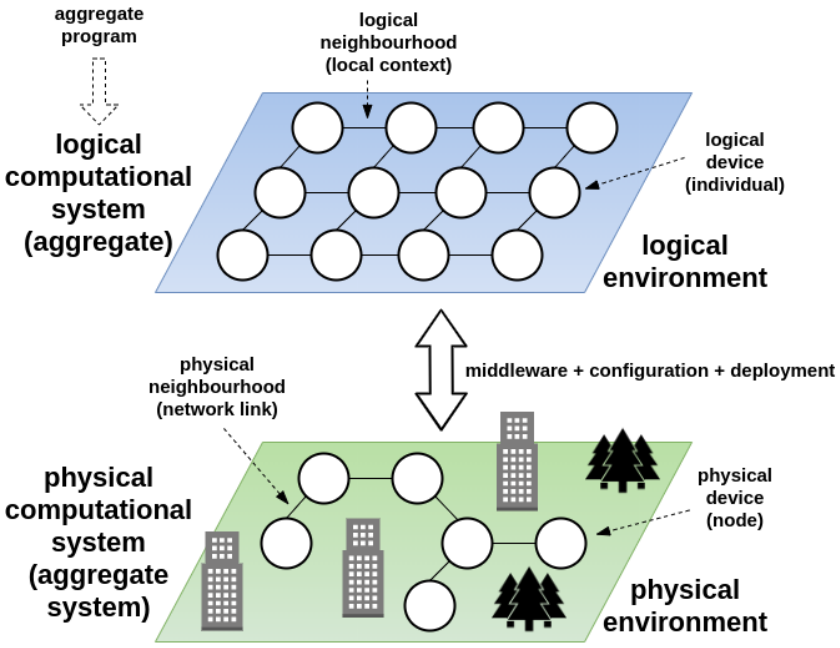


Figure 9.1: Analysis of aggregate systems: logical vs. physical elements.

- A set of *individuals* participating in the same aggregate application is called an *aggregate* or *ensemble*.
- Individuals have an *identity* and are *situated* into some *environment* that can be inspected through *senses* and acted upon through *effectors*.
- An individual can interact with other individuals, which form what is known as its *neighbourhood*, by exchanging *messages*.
- *Abstraction principle*: aggregates and their components (individuals) are logical entities that may be mapped diversely to *physical machines* (machines corresponding to individuals are usually called *nodes* or *devices*). The set of physical devices that sustain the execution of an ensemble is sometimes called an *aggregate system*. The communication technology used for exchanging messages is also abstracted. Figure 9.1 graphically shows the distinction between logical and physical aspects.

9.1.2 Logical analysis

When does an aggregate springs into existence? It depends on the desired notion of "bootstrap": it might be when one or more individuals start to play an aggregate contract, or as soon as the aggregate is *institutionalised* (e.g., by registering it in some *organisation*). In general, multiple, distinct aggregates with the same aggregate contract may (co-)exist; so, any aggregate should have some kind of *aggregate identity*, represented by some *unique identifier (UID)*.

Individuals might exist independently of an aggregate, as parts of some organisation. The individuals that participate to an aggregate are called *participants*. Individuals may *join* or *quit* an aggregate at any time, as regulated by the aggregate contract. Any individual has an *UID* within any aggregate it participates to, to distinguish itself and other individuals from yet other individuals.

9.1.3 Analysis: aggregate execution

The field calculus framework defines, through its operational semantics, how field computations are to be locally executed. This description is abstract: it leaves many details out, allowing for different concrete *execution protocols* or *strategies*. It is important to notice, however, that certain details of execution may be important for a given aggregate application. As a consequence, the middleware should provide the means for *configuring execution-related aspects*. Also, in general, certain details may be enforced by the middleware or (fully or partially) delegated to the device; for instance, the middleware might provide a value range for the firing frequency, and devices might autonomously choose when to execute a round (and a policy should say what happens when a device does not respect the "aggregate contract").

By the most abstract perspective, as illustrated in Figure 9.2, an aggregate application carries out an information-based process for "coherently" turning aggregate-level input to aggregate-level output. In principle, even the deconstruction of an aggregate into individuals could be decided by the platform; in this view, individuals become "contexts" for probing and acting upon some environment.

Macro, execution-related aspects for an aggregate application include the following:

- *Individual context* — In general, the *context* of an individual is given by the

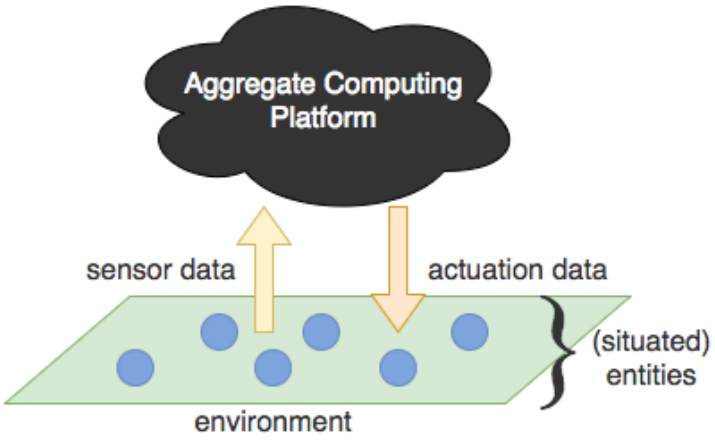


Figure 9.2: Aggregate applications: high-level perspective.

part of the environment it can access, its neighbours, and its state. So, the context provides sensory data, neighbourhood data, and state as input; also, an individual can act upon the context through effectors, communication, and storage acts.

- *Computations of individuals* — Execution proceeds in *computational rounds*. In a computational round, an individual locally runs the aggregate program against its context (the individual is said to *fire*). Also, in general, the program describes how the individual should act upon its context. Certain operations, like the propagation of coordination data (called an *export*) to neighbours, are typically *implicit*, while other (e.g., commands to effectors) explicit.
- *Scheduling of activities* — When does an individual fire? In general, an individual may autonomously choose when to fire, or may be said to do so by an external entity. In any case, the (internal or external, local or global) entity that triggers the execution of the aggregate program for an individual is called a *scheduler*. A scheduler may use local or global context knowledge to decide or adjust the actual scheduling strategy. The scheduler may also dictate when context-related activities are to be performed.

Notice that these descriptions are quite abstract and point out how concerns could be typically handled.

A key operational aspect for aggregates is context management. It comprises

handling communication with neighbours in order to steer and coordinate collective decision-making. The problem of interaction among individuals includes the following issues:

- *Neighbour discovery* — Some mechanism has to be used for devices to be able to get acquainted with one another.
- *Neighbour communication* — Communication may happen in three main modalities: *(i) push*, where an individual sends a message directly to a neighbour; *(ii) pull*, where an individual asks a neighbour for messages; or *(iii) publish/subscribe*, where an individual *emits* information with no particular recipient and listens to relevant information in the environment.
- *Message authentication* — Messages should not be counterfeit, and it should be possible to verify the source of a message.

9.2 ScaFi Platform: Design and Implementation

9.2.1 Situated actors abstraction

When it comes to design and implement distributed systems composed of multiple autonomous entities, the *actor model* [Agh86] is generally considered a primary choice, for it captures the key aspects there involved: distribution, encapsulation of control, and asynchronous communication. Indeed, several works proposed actor-based abstractions and frameworks in the context of particular distributed computing scenarios, such as wireless sensor networks (WSNs) and mobile ad-hoc networks (MANETs) [Bea+13; Ni+05], as well as the IoT, for both middleware [Ngu+17] and application [HCS14] development. In IoT frameworks, for instance, physical devices and services can be wrapped or exposed as actors [HCS14; Lat+15] in order to promote application integration [Lat+15; PA15], behaviour compositionality [Lat+15], and runtime adaptation [VCP16; Val+08; PA15].

To more stress the notion of (physical) *environment*, it is then natural to consider actors for situated systems, which we shall call *situated actors*. However, actor-based applications that involve complex coordination among several (potentially myriads of) entities, and requiring system adaptivity and resiliency, are still very difficult to build: development and maintenance tend to become con-

voluted and brittle due to the scattering of multiple concerns across many actor definitions and intricate conversational patterns [BPV15]. Plausibly, this problem can be addressed by augmenting the actor model with effective abstractions for programming complex collective adaptative behaviours, providing resiliency and support for very-large scale sets of situated components somewhat inherently.

In [CV18], we draw a bridge between the actor model and aggregate computing, in order to establish a disciplined approach for the injection of self-adaptive and advanced coordination capabilities in complex distributed applications. On the one hand, we propose the idea of viewing aggregate computing as a layer on top of actors that enables effective specification of complex coordination patterns. Namely, we describe an actor-based programming framework in which large sets of actors responsible for complex coordination, which we call *actor aggregates*, are programmed "in one shot" according to the aggregate computing model, to automatically and transparently interact with each other to carry on a complex computational process over space and time. With respect to traditional actor programming techniques, this approach reduces accidental complexity by fostering declarativity, separation of concerns, and modularity. On the other hand, our work suggests that a careful exposure of the actor-based view of an aggregate system can provide the means for *(i)* steering collective computation by the inputs of other non-aggregate subsystems of actors, and for *(ii)* turning the aggregate process into coordination events forwarded to the many different parts of a larger application. In a nutshell, integrating aggregate computing *on top of (as well as aside to)* actors is expected to pragmatically address open challenges in the state-of-art of IoT and CASs, by proposing a principled way to the engineering of (critical portions of) such systems.

Actors in the pervasive cyberspace An *actor* [Agh86] is a (re)active entity that represents an independent *locus of control*, encapsulates a state and behaviour, has a globally unique immutable identifier that allows for location transparency, and interacts with other actors via asynchronous message passing—each actor has a *mailbox* for message buffering. In response to a message, an actor can only (i) send a finite number of messages to other actors, (ii) create a finite number of child actors, and (iii) change its own behaviour, that is, its message processing logic.

Thus, an actor system consists of an (possibly huge) evolving set of (possibly changing, mobile) autonomous actors that communicate with one another and perform some task along the way.

Actor systems very well fit highly distributed systems: since communication is based on logical identifiers, the programmer can ignore the actual physical location in which a recipient actor resides (*location transparency*). Also, actors do not share state, in that they communicate exclusively by exchanging messages; as a consequence, the issues related to lock-based synchronisation and mutual exclusion – as found in thread-based concurrency – are completely avoided. In addition, an asynchronous communication style better captures the way in which events occur and are perceived in the physical world [Arm07]; anyhow, synchronicity (sometimes suitable when programming) can be supported as a particular case [Agh86].

Given the appropriateness of actors for modelling distributed systems, it comes naturally to consider them when approaching the development of particular kinds of modern distributed systems, such as large-scale situated systems and those found in the IoT scenario [HCS14]. Here, the *environment* abstraction becomes prominent and paves the path to *context-awareness*, namely, the ability of distinguishing situations depending on context, which is a peculiarity of any "intelligent" behaviour and is often linked to a *locality* principle, i.e., an entity is mostly directly affected by its immediate (logical or physical) surroundings (which effectively represents its context of operation). In this frame, we can introduce a notion of *situated actor* as the bridging abstraction that adds support for *situatedness* on top of plain actors. That is, a situated actor is an actor that has a given position in an environment or generally in space-time—position often inherited by the hosting or associated physical node. Concretely, such actors can be the software interface to a sensor, an actuator, a processor, or generally a computational *device* immersed in some environment, such as the urban area of a smart city, the elevator of a smart building, a room in a smart house, or an edge part of a smart appliance. In other words, a situated actor can be seen as an *avatar* [Mri+15; Zam17] or *digital twin* [ES18] for a physical device, and most specifically, what we can call a *space-aware avatar*.

Starting from this viewpoint, we focus on how actors could be used to implement complex decentralised behaviours, possibly involving a very-large scale

set of devices (and hence actors), as those found, for example, in contexts such as crowd engineering, smart mobility, swarms of drones, environment monitoring, and so on. In principle, this would require a design of the actor system which takes into account discipline, best practices, well-known messaging [Ver15] and structural/behavioural/reactive design patterns. However, when the logic to be expressed involves multiple concerns along different dimensions and abstraction levels, the development and maintenance processes might turn out to be very complicated and costly. There are, in fact, certain system-level properties and algorithms that are difficult to implement when reasoning in terms of individual actors and conversation patterns between actors.

What abstractions could be added to the standard actor model for addressing issues ranging from system-level adaptivity and resiliency to decentralised computation design? How could we build actor-based applications in terms of the *composition* of primitive services (e.g., reusing a crowd estimation service to develop both driving congestion-aware navigation and dispersal advice)? Following the principle of separation of concerns, we could address each problem with the more appropriate paradigm, yet importantly, recovering compatibility between the different views to provide a coherent framework for building complex adaptive systems.

9.2.2 Architectural styles

An *architectural style* characterises a class of systems through a pattern of structural organisation based on a vocabulary of *components*, *connectors*, and *constraints* [GS93]. Accordingly, the ScaFi platform can provide support for organising systems around different architectural styles.

P2P actor-based The most natural system architecture reflects the logical, spatial model of aggregate programming, where devices interact with each other on a local basis—in a fully decentralised manner.

In general, a device actor can be thought of as composed of multiple (child or attached) actors, each one assigned with a single, specific responsibility. Figure 9.3 depicts a complete (i.e., fully operational) device. Sensor and actuator actors handle interaction with the environment. If the device computes on-site, it has

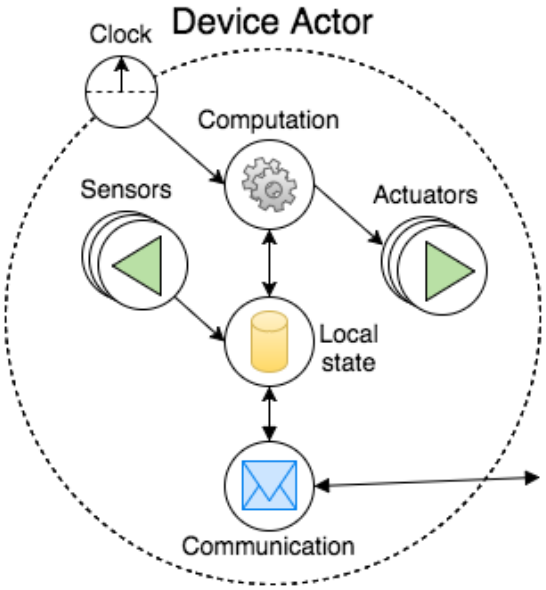


Figure 9.3: Conceptual model of a device actor in ScaFi: each concern (computation, sensing, actuation, state store and communication) is handled by a different child actor. Different platforms may move some child actors outside the device.

a computational actor, which is triggered by clock signals as sent by an internal or external entity. Such a computation actor queries the state actor for inputs—which include the result of the previous computation, the local sensor values and the exports of neighbour devices. Finally, the communication actor is responsible for getting exports from the neighbourhood and propagating the result of each computation round nearby.

In the peer-to-peer platform style (Figure 9.4), concretised by the `BasicActorP2P` incarnation, the system is a network of devices (nodes) represented by actors that, at each scheduled round of execution, compute the aggregate program and broadcast the result message to their neighbourhood. At this level, we abstract from the way the neighbourhood set is discovered: for example, it may be given at configuration time or provided by a neighbouring sensor.

Figure 9.6 shows how a programmer can easily start a node with a default configuration.

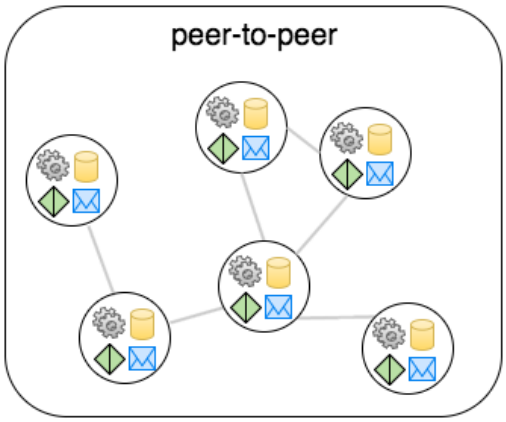


Figure 9.4: ScaFi platform: peer-to-peer style.

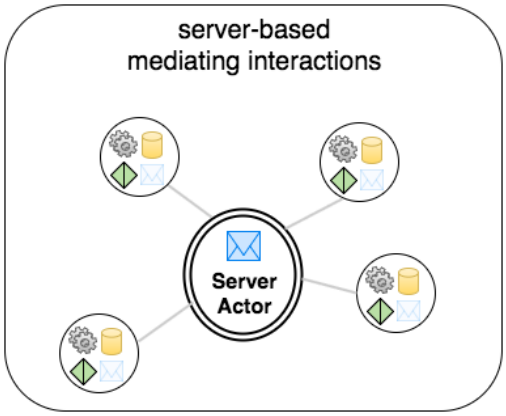


Figure 9.5: ScaFi platform: server mediating interactions.

Actor mediating interactions In some cases, it may be useful to move some duties of the devices to a central entity: an actor that can provide system-wide services and encapsulate environmental features.

As a first example, a server can mediate all device communications by keeping a representation of the space in which they are situated (which may be purely logical or a representation of the physical space and situation), and hence use a configurable distance metric to reify an application-specific notion of neighbourhood for each device. In the crowd steering settings, one such server would easily overcome the difficulties of smartphone local interactions (e.g., via Bluetooth).

Figure 9.5 illustrates one such platform style that is based on a central actor working as a mediator for device-to-device communications. This new platform

```
// STEP 1: CHOOSE INCARNATION
import scafi.incarnations.{ BasicActorP2P => Platform }
import Platform.{AggregateProgram, Settings, PlatformConfigurator}

// STEP 2: DEFINE AGGREGATE PROGRAM
class Program extends AggregateProgram  with CrowdSensingAPI {
  def main() = dangerousDensity() // Specify aggregate computation
}

// STEP 3: PLATFORM SETUP
val settings = Settings()
val platform = PlatformConfigurator.setupPlatform(settings)

// STEP 4: NODE SETUP
val sys = platform.newAggregateApplication()
val dm = sys.newDevice(id = Utils.newId(),
                       program = Program,
                       neighbours = Utils.discoverNbrs())
```

Figure 9.6: Setup of a node in the P2P platform style.

is essentially obtained by a simple reallocation of responsibility, where the communication burden (and knowledge of neighbours) is moved from each device's communication actor to a single server's actor, receiving local computation results and sending the neighbourhood state.

Actor mediating computations In another scenario we may move the aggregate computation from devices to the central server: the devices collapse to system sensors and actuators, essentially becoming environmental contexts upon which the aggregate system can perceive and act. The situation is represented by Figure 9.9. Computation, state management, and neighbourhood communication responsibilities move from device actors to the central actor, which uses a (persistent or in-memory) database to store the global field.

This approach could provide a number of benefits: devices could be unaware of the actual aggregate program to run (which can then be modified on-the-fly in the server), and global optimisation techniques could be adopted to avoid computing all rounds in all devices.

```
// STEP 1: CHOOSE INCARNATION
import scafi.incarnations.{ BasicActorServerBased => Platform }
... // STEP 2,3,4 as in the P2P version
dm.addSensorValue(name = Utils.LocationSensorName,
                  provider = ()=>Utils.getLocation())
```

Figure 9.7: Setup of a node in the platform style based on a mediator of interactions. The code is mostly the same as in the P2P case. The selection of a different ScaFi incarnation gives a new semantics to all the above method calls. In addition, we need to configure a location sensor providing device position to be sent to the server.

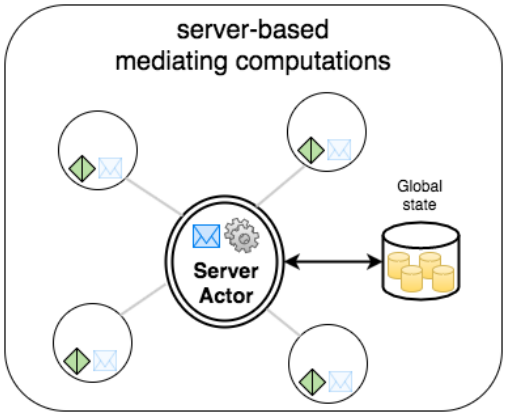


Figure 9.8: ScaFi platform: server mediating computations.

Mixing actor mediating/computing We have seen so far that a plausible execution architecture for aggregate systems can be based on a centralised entity which can, for example, be implemented as an actor. This server can be in charge of locality-based information propagation or computation. A possible next step is to envision a server which dynamically switches between merely mediating communications or computing aggregate programs.

The key insight of this chapter lies exactly in the independence of an aggregate computation from the underlying execution strategy. In fact, thanks to its "pulverisation" semantics, an aggregate computation can be ultimately performed at the device site or by a computing entity that is able to correlate global and local information.

There may be practical reasons to opt for centralised execution platforms—

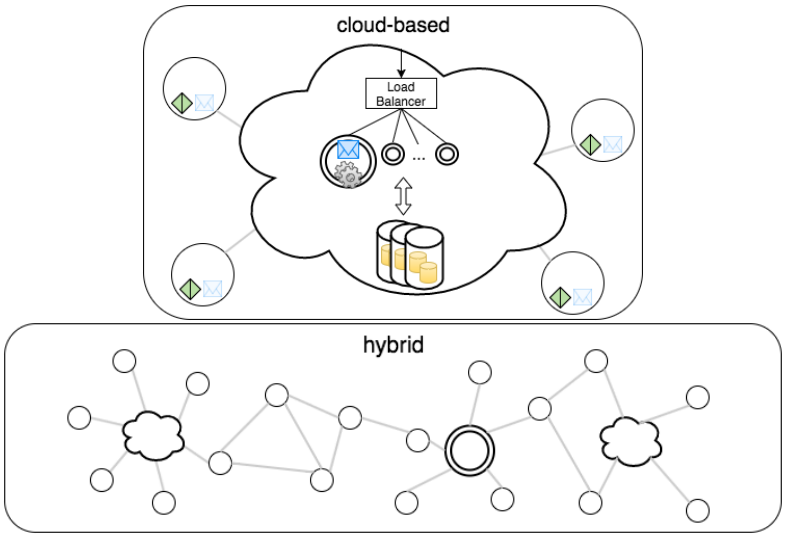


Figure 9.9: ScaFi platform: cloud and hybrid styles.

e.g., for easier maintenance, to enforce security policies, or because broadcasts to neighbourhoods are not supported at the infrastructure-level. Secondly, the generality and abstractness of aggregate computing can provide greater flexibility with respect to *how* the ensemble ultimately carries out computations. This means that (i) the platform can make the best use out of the computational and networking resources at hand, and (ii) it can opportunistically adapt the execution strategy to changes in the available environment or computational infrastructure. It is possible to reason in terms of movable or fluent responsibilities, in the sense that certain operations can "flow" from devices to computing servers, or vice versa—dynamically.

Aggregate Computing in the Cloud The introduction of central servers seems to contradict the original purpose of the aggregate computing approach, which fits fully decentralised distributed computing scenarios. However, handling large numbers of devices is possible using cloud-oriented approaches.

Cloud computing is a well-established model and technology supporting scalability and elasticity through on-demand provisioning of IT resources—which are typically virtualised. Since it represents a further opportunity for building scalable systems, it is reasonable to think of an alternate execution strategy for aggregate systems where computations are carried out in the cloud.

A main strategy for a cloud-based execution platform consists in *storing the whole computational field as a big data*, with aggregate computation structured as a myriad of stateless computing services concurrently working on a big shared database.

The global aggregate computation might even be executed partially "on ground", e.g., as advocated in edge- and fog-computing initiatives [Bon+12], and may "flow" up and down depending upon context and contingencies—energetic issues, presence of congestions, unexpected storage requirements, changes in wireless availability, and so on.

9.3 Final Remarks

This chapter covers aspects related to providing middleware-level support for aggregate computing, and presents the ScaFi platform as a proof-of-concept implementation. Supporting different architectural styles and execution strategies is the fundamental, starting point for more ambitious goals, such as that of enabling dynamic, opportunistic adaptation of the execution strategy of aggregate systems, depending on multiple factors ranging from available infrastructure and quality-of-service requirements. Indeed, a middleware is the place where a wide range of optimisations may be applied. However, developing a middleware is a significant engineering challenge: further work is needed to move this proof-of-concept implementation into a full-fledged, flexible, optimised, production-ready framework with high-quality API.

References

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA, USA: MIT Press, 1986. ISBN: 0-262-01092-5.

[Arm07] Joe Armstrong. *Programming Erlang: software for a concurrent world.* Pragmatic Bookshelf, 2007.

[Bea+13] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. "Organizing the Aggregate: Languages for Spatial Computing". In: *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. A longer version available at: http://arxiv.org/abs/1202.5509. IGI Global, 2013. Chap. 16, pp. 436–501. ISBN: 978-1-4666-2092-6. DOI: 10.4018/978-1-4666-2092-6.ch016.

[Bon+12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. "Fog computing and its role in the internet of things". In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM. 2012, pp. 13–16.

[BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. "Aggregate Programming for the Internet of Things". In: *IEEE Computer* 48.9 (2015), pp. 22–30. DOI: 10.1109/MC.2015.261.

[CV18] Roberto Casadei and Mirko Viroli. "Programming Actor-Based Collective Adaptive Systems". In: *Programming with Actors: State-of-the-Art and Research Perspectives*. Vol. 10789. Lecture Notes in Computer Science. Springer, 2018, pp. 94–122. DOI: 10.1007/978-3-030-00302-9_4.

[ES18] Abdulmotaleb El Saddik. "Digital twins: The convergence of multimedia technologies". In: *IEEE MultiMedia* 25.2 (2018), pp. 87–92.

[GS93] David Garlan and Mary Shaw. "An introduction to software architecture". In: *Advances in software engineering and knowledge engineering*. World Scientific, 1993, pp. 1–39.

[HCS14] Raphael Hiesgen, Dominik Charousset, and Thomas C Schmidt. "Embedded Actors – Towards distributed programming in the IoT". In: *2014 IEEE Fourth International Conference on Consumer Electronics Berlin (ICCE-Berlin)*. IEEE. 2014, pp. 371–375.

[HM06] Salem Hadim and Nader Mohamed. "Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks". In: *IEEE Distributed Systems Online* 7.3 (2006). DOI: 10.1109/MDSO.2006.19. URL: https://doi.org/10.1109/MDSO.2006.19.

[Lat+15] Elizabeth Latronico, Edward A. Lee, Marten Lohstroh, Chris Shaver, Armin Wasicek, and Matthew Weber. "A Vision of Swarmlets". In: *IEEE Internet Computing* 19.2 (2015), pp. 20–28. ISSN: 10897801. DOI: 10.1109/MIC.2015.17. URL: https://cloudfront.escholarship.org/dist/prd/content/qt7p53t9x5/qt7p53t9x5.pdf.

[Mri+15] Michael Mrissa, Lionel Médini, Jean-Paul Jamont, Nicolas Le Sommer, and Jérôme Laplace. "An avatar architecture for the web of things". In: *IEEE Internet Computing* 19.2 (2015), pp. 30–38.

[Ngu+17] Anne H Ngu, Mario Gutierrez, Vangelis Metsis, Surya Nepal, and Quan Z Sheng. "IoT middleware: A survey on issues and enabling technologies". In: *IEEE Internet of Things Journal* 4.1 (2017), pp. 1–20.

[Ni+05] Yang Ni, Ulrich Kremer, Adrian Stere, and Liviu Iftode. "Programming ad-hoc networks of mobile and resource-constrained devices". In: *ACM SIGPLAN Notices* 40.6 (2005), pp. 249–260.

[PA15] Per Persson and Ola Angelsmark. "Calvin–merging cloud and iot". In: *Procedia Computer Science* 52 (2015), pp. 210–217.

[SS02] Richard E Schantz and Douglas C Schmidt. "Middleware". In: *Encyclopedia of Software Engineering* (2002).

[Val+08] Jorge Vallejos, Elisa Gonzalez Boix, Engineer Bainomugisha, Pascal Costanza, Wolfgang De Meuter, and Éric Tanter. "Towards Resilient Partitioning of Pervasive Computing Services". In: *Proceedings of the 3rd Workshop on Software Engineering for Pervasive Services (SEPS 2008)* January 2008 (2008), pp. 15–20. DOI: 10.1145/1387229.1387234. URL: https://www.researchgate.net/profile/Eric{_}Tanter/publication/234802434{_}Towards{_}resilient{_}partitioning{_}of{_}pervasive{_}computing{_}services/links/02e7e52cc526b22fd7000000.pdf.

[VCP16] Mirko Viroli, Roberto Casadei, and Danilo Pianini. "On execution platforms for large-scale aggregate computing". In: *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. ACM. 2016, pp. 1321–1326.

[Ver15] Vaughn Vernon. *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka*. 1st. Addison-Wesley Professional, 2015. ISBN: 0133846830, 9780133846836.

[Zam17] Franco Zambonelli. "Key Abstractions for IoT-Oriented Software Engineering". In: *IEEE Software* 34.1 (2017), pp. 38–45.

Chapter 10

Self-Organising Coordination Regions: a Pattern for Edge Computing

Complexity is looking at interacting elements and asking how they form patterns and how the patterns unfold.

W. Brian Arthur

Contents

Aggregate computing provides perspectives and inspiration for techniques and approaches to distributed system design. This chapter presents a general, decentralised coordination design pattern for partitioned orchestration which was discovered from multiple occurrences in the form of a "S-G-C-G" chain of aggregate building blocks (see Sections 5.2.2 and 7.5.1). It aims to provide adaptivity and resilience in large-scale situated systems through multiple feedback loops involving dynamic system partitions. The pattern is called ***Self-organising Coordination Regions (SCR)***, since it works through an internally-regulated, adaptive construction of regions where activity is coordinated via feedback/control flows among master and worker nodes. In other words, it leverages asymmetry in complex coordination scenarios and accordingly proposes a tunable trade-off between centralised and decentralised decision-making.

The described pattern finds application in several scenarios where a sparse set of leaders is expected to collect feedback from and enact decisions for a subset of other participants—examples include target counting [PDV17], group management for target tracking [Liu+04], decentralised service orchestration [JDB16], self-adaptative software [Wey+13], Wireless Sensor Networks (WSN) [DRT05; LRM08], robot swarm control [WAC+14], crowd tracking and steering [BPV15].

The rest of this chapter is structured as follows, with content following roughly the GoF pattern template form [Vli98]. Section 10.1 provides motivation, context and discusses related work and patterns. Section 10.2 presents the pattern by providing its intent, synonyms, structure, dynamics as well as known uses, consequences and methodological guidelines of its application. Section 10.2.6 shows an implementation in the Aggregate Computing framework, and discusses variants. Section 10.3 provides empirical evaluation. Finally, Section 10.4 provides some concluding thoughts.

10.1 Motivation

10.1.1 Need for design patterns for self-* systems

Design Patterns are paramount in software engineering. They capture expert knowledge by describing reasoned solution schemas for a well-defined class of repeatedly occurring problems in specific contexts [Bus+96]. Patterns help to harness complexity by characterising systems of forces arising in a context, and strategies to resolve them [Ale77], while abstracting from implementation details, denoting intents and properties of solutions, providing motivated guidance towards desired configurations, and supporting documentation and team communication through a common vocabulary [Bus+96]. Over time, several classes of patterns have been discovered to assist designers and implementors of software-based systems, resulting in *catalogues* of patterns, e.g., for language implementation [Par09], object-oriented software [Vli98], concurrency [Sch+00], messaging [HW04], reactive systems [Ver15; KHA17], asynchronous programming [Cas16], fault-tolerant software [Han13] etc. Moreover, patterns can be classified into multiple taxonomies (e.g., by level of abstraction into architectural, design patterns, and idioms [Bus+96]), can be related to each other (e.g., by refinement, variance, and combination [Bus+96]), and can be presented using different formats (e.g., *Alexandrian* [Ale77], *GoF* [Vli98], and *POSA* [Bus+96]).

10.1.2 Context

In this chapter, we consider the context of coordination in large-scale distributed systems. Specifically, we focus on scenarios – e.g., pervasive computing, Collective Adaptive Systems (CAS), Internet of Things (IoT), Cyber-Physical Systems (CPS), and Edge Computing – characterised by the following forces:

- *Distribution.* Having distributed components leads to concurrency, lack of global clock, and independent (and often frequent) failure or unavailability of components [CDK05]—with corresponding implications.
- *Cyber-physicality.* The system may consist of both disembodied and physically embedded components.

- *Situatedness.* Components may be logically or physically immersed into an environment such that their location and context are relevant, since their inputs and outputs may be limited to the surroundings.
- *Heterogeneity.* Components may differ by their computational capabilities, energy requirements, and general dependability.
- *Large scale.* Systems may be too large to be centrally orchestrated or manually operated.

Given the rather intense research ongoing in these contexts, their broad scope, complexity of the challenges, and proliferation of paradigms, some catalogues of design patterns have emerged. Relevant examples include pattern catalogues for multi-agent architectures [HCY+99] and ensemble structures [HL04], bio-inspired computing [FM+13; Bab+06], and decentralised control [Wey+13] and coordination [DWH06] in self-adaptive systems. They typically work at different levels of abstractions, from principles and high-level behaviour components to mathematically-defined evolution rules, and do not generally provide complete solutions for the complex problem of scalable coordination of large-scale situated systems.

Edge computing (Chapter 6) is a motivating scenario for SCR.

10.1.3 Problem and forces

Most specifically, in such scenarios, the *problem forces* that must be dealt with include the following:

- *heterogeneity* creates asymmetry in individual capabilities, or tasks are so complex that *collaboration* is essential, e.g., the information, rights, or resources available at an individual device are not sufficient for it to autonomously carry out the task at hand;
- a *locality principle* holds, as context is key for both individual and collective activity, and cost is typically proportional to the distance between sources, processes, and users;
- neither *full centralisation* nor *full decentralisation* in control and decision-making is possible or desirable—the former for evident scalability reasons,

the latter for the inherent complexity in achieving consensus and globally optimised functions; and

- the environment and system structure are *dynamic* (e.g., due to emergence of events that must be dealt with, mobility or failure), creating a situation of constant change where the system stability is continuously endangered by perturbations.

10.1.4 Basic patterns and abstractions

The SCR pattern, described in detail in Section 10.2, basically consists of a subdivision of the system into regions regulated through feedback-and-control loops between leaders and the other agents. It recurs in a number of scientific works and proposed solutions, and is implemented variously. In this section, its component patterns are introduced.

Some patterns presented in the aforementioned catalogues [FM+13; Vir+18] constitute the foundations of the current work. Indeed, the SCR pattern is a combination of three fundamental coordination (sub-)patterns:

- *Multi-leader election.* In distributed systems, it is sometimes useful to break symmetry or introduce multiple local centralisation points to simplify decision-making or coordination. This pattern consists in the election of multiple leaders to uniformly cover a logical or physical space.
- *Information propagation.* Communication patterns that abstract from low-level implementation or networking details are essential in distributed systems. This pattern consists of propagating information from one or more sources outward, independently of the underlying system structure.
- *Information collection.* This pattern consists of collecting information from a set of sources into one or more sinks, still abstracting from low-level details.

In order to account for situations where devices can fail or change, coherently to the self-organisation principle, we should consider the above patterns as *continuous processes* (or, at least, as processes that are *reactive* [MC14] to failure or change). This means that information (updates) must move continuously, as a stream (logically, and despite potential optimisations), as captured by the *information flow* abstraction, defined in [DWH07] as follows:

> An information flow is a stream of information from source localities towards destination localities and this stream is maintained and regularly updated to reflect changes in the system. Between sources and destinations, a flow can pass other localities where new information can be aggregated and combined into the information flow.

A common way to implement information flows is by activating processes that create and maintain structures for the communication paths. One such example is the *gradient* [DWH06; Aud+17; DB16], a self-healing distributed data structure mapping any node of the system to its hop-by-hop estimated distance from source points: it provides an underlying carrier for controlling effective directions of propagation/collection of data flows. Information flows can be naturally expressed in the library of [Vir+18], which fosters the definition of collective behaviour of an ensemble of devices through a composition of self-organising patterns, drawing inspiration from biology [FM+13]. The aforementioned sub-patterns are "building blocks" in [BPV15], where are respectively called S (for ***S**parse-choice*—i.e., a scattered selection from the set of participating devices), G (for ***G**radient-cast*—i.e., a multicast diffusing information along a gradient), and C (for ***C**onverge-cast*—i.e., a multicast aggregating information to a sink device).

10.1.5 Related patterns

A well-known organisational meta-pattern for self-adaptive systems is *MAPE* [KC03]: it suggests structuring the system feedback control loop into four components: *Monitor*, *Analyse*, *Plan*, and *Execute*. In [Wey+13], several MAPE patterns are provided for organising the adaptation logic in decentralised self-adaptive systems. These are related and operate in a similar design context, but their focus is on internal organisation of system adaptivity rather than on external, application design. In particular, the *Regional Planning* pattern [Wey+13] consists in distributing *Planning* components to different "software regions" (i.e., loosely coupled software subsystems); there, they collect data from *Analyse* components (which are fed by *Monitoring* components) and command *Execute* components for enaction of planned adaptations. SCR subsumes Regional Planning: it enables the design of self-adaptation control loops but goes beyond that, by covering various

assignments of responsibilities to the participants and being directly usable for application logic as well; e.g., leaders in SCR may gather regional data, resolve contention, or propagate events.

The *Multi-Scale Feedbacks* pattern [DDFM19] deals with large-scale coordination in hierarchical self-* systems. The pattern characterises a self-* system as a set of entities, exposing *observable* features, that are *associated with* or *composing* other entities. Then, it defines *micro-to-macro information abstraction* and *macro-to-micro feedback* as key functions between micro and macro features. Even though SCR can be applied to hierarchies and hierarchically, and shares some similarities with Multi-Scale Feedbacks – e.g., *inter-level feedbacks* (*downward/upward causation*) are comparable to SCR downstream and upstream information flows, assuming leaders are at a higher level than other agents –, the two patterns have different goals and take different perspectives: while SCR focusses on coordination of decentralised activity and interactions, Multi-Scale Feedbacks focusses on hierarchical design.

10.1.6 Known Uses

Various forms and uses of the SCR pattern can be found in literature.

Decentralised service orchestration In [JDB16], SCR is used to design a decentralised service orchestration system; there, a workflow specification is split for scalability and performance into sub-workflows executed by multiple collaborating engines that are migrated to different network regions based on placement analysis.

WSN middlewares *TCMote* [DRT05] is designed according to SCR. The system is organised in (possibly hierarchical) *sensor regions* governed by *leaders* with higher capabilities than the other region nodes (called *motes*). TCMote uses tuple channels for one-to-many and many-to-one communication between region sensors and the region leader in a single-hop. In another WSN middleware, *TS-Mid* [LRM08], tuple space-based logical regions are used for power saving; there, regional leaders dispatch operations to normal nodes and transmit results to sink nodes.

Swarm robotics In the swarm steering study [WAC+14], the authors leverage dynamically selected, human-controlled leaders to influence and guide robot swarms towards dynamic goal regions.

Traffic light control The framework [JM18] for decentralised traffic light control is based on hierarchical multi-agent system organised as per SCR. *Region agents* model regions of the traffic network. They consist of *intersection agents* (SCR leaders) that coordinate with other intersection agents and control a set of *turning movement agents* (SCR downstream process), which learn to behave collectively and provide feedback at the corresponding intersection (SCR upstream process).

Resource management In [YKO03], a hierarchical system for the management of resources in large-scale multi-agent domains is described. In this approach, called *Distributed Dispatcher Manager (DDM)*, agents are organised into teams where communication is restricted to happen only between group subordinates to the corresponding team leaders. Leaders, which can also be grouped to form higher-order teams. collect information from subordinates and propagate resource assignments back. In another work, *Mission-oriented Adaptive Placement (MAP)* [Pau+19], SCR is adopted to implement a resource management framework for self-adaptive dispersal of computing services in multi-layer infrastructures. The approach leverages regional load balancing and inter-region coordination for global load balancing.

Decentralised reinforcement learning In [ZLA10], a decentralised approach to reinforcement learning is proposed that leverages a *multi-level, supervision-based organisation* to coordinate the learning process: there, lower-level agents (called *subordinates*) are grouped into clusters, depending on how much they interact together, and report states and rewards to supervising agents (called *supervisors*), which in turn provide supervisory information to guide the learning agents in the exploration of their state-action spaces.

Morphogenesis In [Zah19], SCR is used to implement an algorithm, inspired by the working of vascular systems of plants, for the dynamic distribution of re-

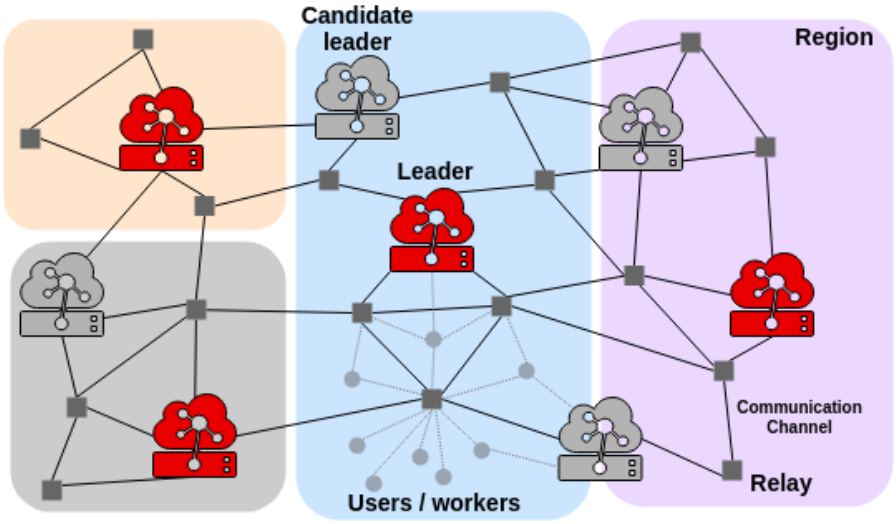


Figure 10.1: SCR from a structural perspective—see description in Section 10.2.1. Notation: "gateway-like" nodes denote candidate leaders (red for active ones, grey for unelected ones); small grey squares denote relays; small grey circles denote users/workers.

sources aimed at the regulation of morphogenetic processes. This is called *Vascular Morphogenesis Controller (VMC)*. A VMC system consists of an acyclic directed graphs where root notes (i.e., the leaders) acquire and distribute resources to leaves in a *forward flow*, and leaves, depending on their environmental conditions, provide a *backward flow* of guiding signals used to adjust the thickness of connections—influencing the amount of resources flowing in, which in turn affects creation and removal of nodes (cf., branching and shedding in plants).

Other known uses Instances of the pattern can be found in other works that include distributed sensing [CV18], target counting [PDV17], group management for target tracking [Liu+04], situated problem solving [Cas+19b], design of self-adaptation control loops [Wey+13] (as discussed above), crowd tracking and steering [BPV15; Cas+19a] in opportunistic IoT, as well as peer-to-peer clouds [CV19].

10.2 SCR Pattern Description

Intent Support scalable control and monitoring of a distributed system, with resiliency to failures and dynamicity, and balancing centralisation and decentralisation in decision-making. In particular, it promotes the formation of dynamic groups of components, while taking into account the context (as induced by the environment or problem space).

Name and synonyms

- *Self-organising Coordination Regions.* This reflects the decentralised nature of this pattern, as well as its support for coordination through scoped, endogenous, emergent structures and dynamics.
- *Decentralised Multi-Orchestration.* This is also a suitable name, as the pattern defines a decentralised coordination strategy for injecting multiple orchestration points into a system, creating corresponding system partitions regulated through feedback loops.
- *SGCG.* This name denotes the chain of aggregate programming blocks that provides a possible implementation schema of the pattern (see Section 10.2.6).

10.2.1 Structure and participants

Structurally, the pattern is organised as of Figure 10.1. The system can be logically represented as a network of *nodes* on which spatially extended and dynamic structures, called *regions*, emerge, each "containing" a subset of devices. These components can assume at any time one or more of the following roles[1]:

- *Candidate leader*: a node that is eligible, by virtue of its position, resources, or capabilities[2] for being elected as leader of a group of nodes or a region of space;

[1]Depending on the scenario and the particular instantiation of the pattern, the types of entities involved may take specialised names, such as those reported in Table 10.1.

[2]Even though the pattern itself makes no assumption on the network structure, on an edge deployment usually candidate leaders correspond to edge servers.

Pattern term	**Synonyms/specialised terms based on context**				
	Networks	*Master/Worker Architecture*	*Cluster Management*	*Coordination*	*Others*
Leader	Hub, Root	Master, Control plane	Manager	Orchestrator, Coordinator	Principal, Supervisor
Candidate leader		Secondary master	Backup manager		
Member	Node	Worker, Slave	Agent	Component, Coordinable	User, Follower, Subordinate, Participant
Intermediary	Relay, Link, Router	Work queue		Channels, Connectors	Forwarder, Intermediary
Region	Partition	Subtask	(Sub-)Cluster	Team, Coalition	Area, Division

Table 10.1: Examples of specialised terminology for the pattern components in different contexts.

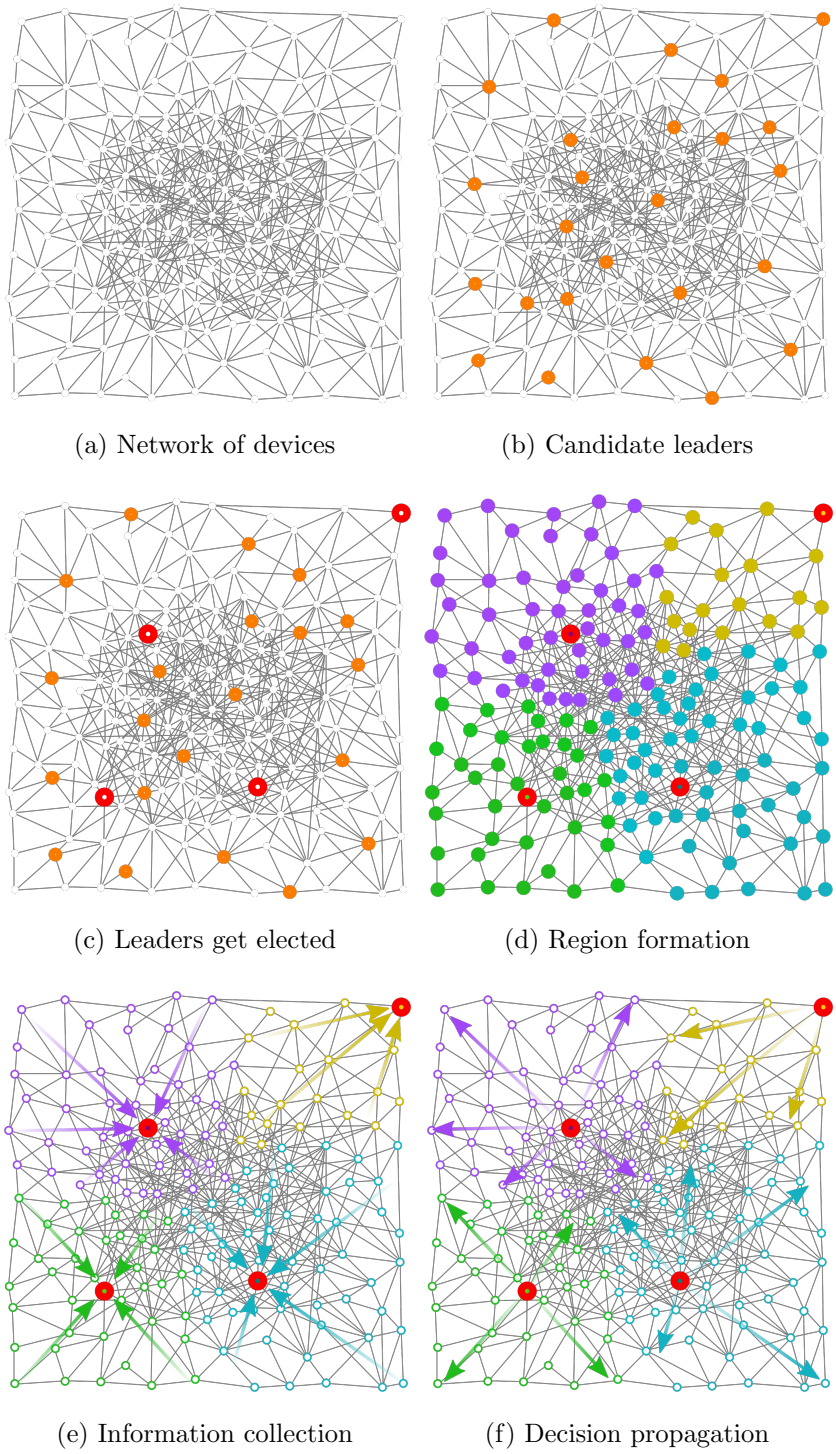
- *Leader*: a node that is responsible for processing information obtained from other nodes in its region and enacting decisions within the region;
- *Member* or *subordinate* (of a region): a node (e.g., a user or worker node) that sends/receives information to/from the leader of the region it is part of, through intermediaries;
- *Intermediary*: a node that mediates interaction between leaders and members.

The regions may be logical or physical, may cover a part or the entirety of the space, and may be strictly separated or overlapping. The intermediaries mediate the interaction between leaders and members; sometimes, e.g., in peer-to-peer networks, these may work as relays.

10.2.2 Dynamics and collaborations

The pattern induces a computational behaviour organised in four phases (Figure 10.2):

1. *Election of leaders.* Leaders are elected from the set of candidates.
2. *Formation of regions.* Structures are created such that each user is assigned to a single leader, and information can flow in both directions through proper communication paths.



(a) Network of devices (b) Candidate leaders

(c) Leaders get elected (d) Region formation

(e) Information collection (f) Decision propagation

Figure 10.2: Series of snapshots showing the phases of the pattern.

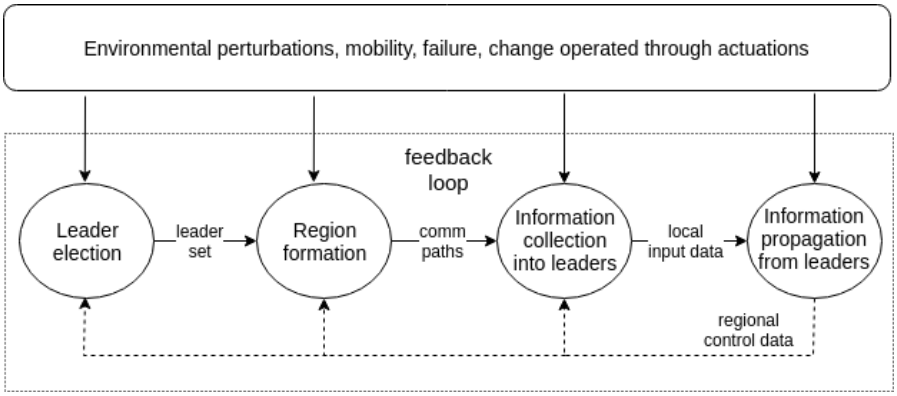


Figure 10.3: SCR from a dynamical perspective—see description in Section 10.2.2. Notation: solid arrows represent required inputs or unavoidable perturbations; dashed lines denote possible feedback loops.

3. *Information flow from users to leaders.* User nodes stream data or updates needed by leaders to achieve the system goals, and some processing can occur *en-route*—examples include sensor data, local events, service requests, or feedback information for the assigned tasks.
4. *Information flow from leaders to users.* Leaders stream computation results to all members of their managed region—it may be a decision to be enacted, a collective view to be propagated, instructions to be assigned, and so on.

Note that these phases are only conceptually sequential: they are rather dynamical processes that happen concurrently, are continuously revised, and are related by input/output dependencies (see Figure 10.3). Specifically, the leader election phase can be thought as an active process black box that can react to various perturbations to automatically revise the selection of leaders and shape of regions; then, as regions change, the corresponding collection and propagation processes need to adapt. Moreover, the system can be configured with feedback loops: information propagated by leaders may produce an effect on workers that can subsequently get perceived by leaders through collected data.

10.2.3 Variants and extensions

- *Leader election with pre-established regions.* In some cases, the regions must be decided before the corresponding leaders are elected.

- *Connected leaders.* In some scenario, communication between leaders is desired to allow for global, system-wide coordination that goes beyond the needs of individual regions.
- *Hierarchical organisation.* The pattern can be applied recursively: a region can be split into sub-regions governed by sub-leaders, and so on.
- *Overlapping regions.* Multiple instances of the pattern may be concurrently spawned with different regions, in order to provide in each device a superimposed view of its various "localities". This requires the capability to execute some parts of the distributed coordination algorithm concurrently.

10.2.4 Applicability

When to apply Use of the SCR pattern is encouraged in any of the following:

- A large-scale situated system needs to self-organise in such a way that its components can be monitored and coordinated according to a view larger than local, such as in complex situation recognition.
- A balance between centralisation and decentralisation is required to support effective decision-making in large-scale, dynamic contexts.
- All or part of the information should be processed nearby the users, because of resource constraints like bandwidth, storage, energy, and so on.
- The underlying network structure is unknown, the system is open (new relays, leader candidates and users can join and leave the system dynamically), failures are possible, or other events can dynamically change the network structure.

When not to apply Adoption of the SCR pattern is discouraged (or would lead to degenerate cases) in the following circumstances:

- Decision-making can be carried out in a fully local way.
- Decision-making must be entirely centralised (actually, this could be tackled by electing a single leader, but more efficient solutions may exist for less dynamic scenarios).
- The network structure is statically defined.

10.2.5 Consequences

The SCR pattern has the following consequences:

- *Hybrid decision-making.* Decisions are taken considering a tunable subset of the whole system, de-facto creating a hybrid between centralised and decentralised decision-making.
- *Sub-network isolation.* Unless an extended version of the pattern is deployed, users belonging to different regions do not participate in the same sub-system (i.e., they do not exchange data).
- *Reduced dependence from deployment and network structure.* SCR creates a sort of dynamic, adaptive network overlay structure on top of the existing communication infrastructure. By merely organising application logic on that overlay, the specific shape of the underlying network can be abstracted away, allowing for easier porting to diverse setups (e.g. cloud, edge, purely P2P).
- *Eventual consistency.* Temporal mobility, loss of messages, and device failures, only temporarily affect the values collected in leaders, and hence, deviation from the actual global view.

10.2.6 Implementation

In this section, we describe some implementation issues and possible variants of the four phases described in Section 10.2.2, and then provide an example specification in ScaFi.

Election of leaders and formation of regions

- *Consensus strategy.* Consensus on leadership may involve centralised algorithms, or resort to (more challenging) algorithms for distributed and asynchronous systems [Sto00].
- *Candidate leaders.* In general, there could be constraints or preferences concerning which nodes can be selected as leaders: coordinators are usually preferably static, dependable nodes with significant computational and network resources, and little or no power saving concern—such as edge gateways

or fog nodes. Trust could also be used to rate and therefore include/exclude nodes from the candidate set based on observed activity.

- *Time of election.* Leaders can be elected statically (i.e., before system execution) or be dynamically reconsidered, continuously or after a delay.
- *Objectives.* The goal is usually a configuration of leaders that must be valid or optimised with respect to a particular property—e.g., uniformity in spatial coverage (as of a smart city environment) or balancing of load (tasks, workers).
- *Adaptivity and resilience.* A new leader election process must be activated when the current leader configuration gets invalidated. E.g., this could happen due to mobility, change of load, or failure of some leader.

Information spreading

- *Gossip.* One way to implement spreading of information is through gossip protocols [Bir07], which are suitable for letting information flow from leaders to users under the condition that the generated information is monotonic (namely, it can only change in a single direction). Whenever such an assumption does not hold, gossip algorithms should get periodically reset (or overlapping replicates of the algorithm should execute in parallel [PBV16]).
- *Gradient-based information cast.* A class of algorithms for distributed information spreading is rooted on the idea of carrying information along with a monotonically-increasing (logical or physical) distance from the information source. This is suitable both for generating regions once leaders are elected (by selecting the closest leader) and for propagating information from leaders to users. Several implementations of the algorithm exist, ranging from distributed adaptive Bellman-Ford [DB16] to advanced versions and compound algorithms taking into account aspects like time, speed, and acceleration of devices [Aud+17].

Information accumulation

- *Gossip.* Information accumulation is generally a tougher task than information spreading. As for spreading, accumulation can be realised by gossiping

information such that the leader is reached with messages from all nodes in the region: however, this effectively works only in the case of small regions.

- *Spanning tree techniques.* A more scalable technique is based on building a spanning tree over the network (locally selecting as parent the closest neighbour to the source), then accumulating along such tree towards the leader. Spanning trees, however, are highly fragile to changes in the network: disruption and creation of links may lead to different configurations, making naive versions of this algorithm unsuitable for mobile scenarios.
- *Multi-path techniques.* Multi-path techniques aggregate information along the source using multiple spanning trees rather than a single one. They are usually more robust to changes in the network structure, but take more time to converge in case of stable networks [Vir+18].

10.2.7 Sample code

We propose an implementation draft for the pattern in the paradigm of aggregate computing [BPV15; Vir+19]—used in next section as a basis for evaluating a smart city case study. The reason for this choice is rooted in the rather straightforward mapping between the sub-patterns of SCR and the building blocks available in existing aggregate computing languages, which allow for a concise implementation.

Pattern implementation schema In ScaFi, the pattern can be encoded as follows[3].

[3]Purple symbols are non-primitive aggregate building blocks, grey symbols are configuration parameters, and bold symbols denote methods for local activity to be tailored to the application.

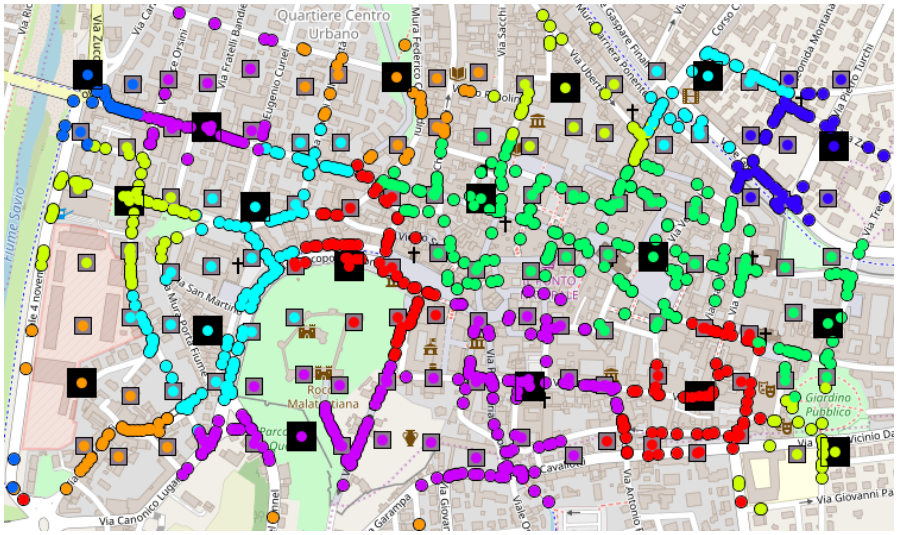


Figure 10.4: A snapshot of the simulation in execution. Edge servers are depicted as square nodes, users as circular nodes. Leaders are black, big squares; unelected leaders (working as relays) are smaller, greyed squares. The colour of the circular dot identifies the id of the region assigned to that node.

```
class SCR extends AggregateProgram with BlockG with BlockC with BlockS {
  def main = {
    // selects a field of leaders, with at least grain distance
    val leader = branch(isCandidate) { S(grain) } { false }
    // creates a gradient from leaders based on a given metric
    val potential = distanceTo(leader, metric)
    // gathers localInput values towards leaders by aggregation
    val convergeCast = C(potential, localInput, aggregationFun)
    // on leaders, takes a local decision based on received data
    val decision = decisionMaking(leader, convergeCast)
    // broadcast decisions and take action
    val divergeCast = G(leader, metric, decision)
    localAction(divergeCast)
  }
}
```

10.3 Evaluation

10.3.1 Case study #1: dynamic area management

In this section, we present an example implementation of the pattern in the context of smart cities and edge computing and evaluate it by simulation to reveal

Name	Description	Values
u	Active user devices count	[50, 100, 200, 500, 1000]
α	Backoff algorithm parameter	[0, 10^{-3}, 10^{-2}, 10^{-1}, 1]
ρ	Probability a leader stops after 10 min	[0, 0.25, 0.5, 0.75, 1]
fb	Flag: feedback loop enabled	[true, false]

Table 10.2: Free variables for the scenario in exam.

its intrinsic self-organisation character.

Motivation Consider a multimedia application that requires computation over user-generated video stream and low-latency communication. Example applications are, e.g., metropolitan collaborative surveillance [Dau+18] and multiplayer gaming. For the latter, pervasive usage of multi-view and 360-degree-view video streams is currently limited by delay intolerance and excessive bandwidth usage [BE17]. Moreover, relevance of low-latency video processing will likely increase in the future with advancements in mobile augmented reality technology [SC12]. One wants such multimedia application to execute on a smart urban environment, where users, equipped with mobile devices (smartphones, or even augmented-reality equipment) can move. The smart city is populated with a network of static (non-mobile) edge servers, with which mobile devices can communicate. The goal is to adaptively select a subset of edge nodes (enough to sustain the computation) to work as local leaders, gather and redirect the video streams from user devices to one leader edge device, process the data gathered, and finally spread the computation result back to the users.

Scenario description We consider a scenario of multiple edge servers (specifically, 126) in the centre of the Italian city of Cesena, all participating in the system as leader candidates. Their positions form an irregular grid, and vary on different simulation runs. We dynamically select a subset of these leader candidates to work as leaders, and let the others participate in the system as relays. More precisely, the edge servers elect a leader for every region of 200 meters in radius, competing using the `S` building block (namely, breaking symmetry using a device local id, and favouring already established leaders if in a proper range).

The goal of the system is to collect data streams generated by users, aggregate

Name	Description	Unit
E of feedback adjustment	Mean of the feedback adjustment for every leader. It measures how much the radius of the coordinated region is extended. Lower values indicate bigger regions.	m
σ of feedback adjustment	Standard deviation of the feedback adjustment for every leader. It is an indication of how much the radius of the coordinated region varies among leaders. Higher values indicate higher disparity in such values, meaning that the feedback system is altering the region sizes more intensively.	m
$\sum$ of clients per edge server	Overall number of users served. The value should ideally match the number of users in the system. Higher values indicate streams being processed by multiple leaders (due to users changing region), lower values indicate non-served users.	users
σ of clients per edge server	Standard deviation of number of users served by each leader. Indication of load balancing. Higher values indicate that more computational capacity is required for some leaders w.r.t. others. The lower, the better balanced is the load.	users

Table 10.3: Dynamic area management: measures for the case study.

it, and diffuse to the whole region the number of streams being processed. Users are modelled as devices moving along roads open to pedestrian traffic (data obtained from OpenStreetMap [HW08]) at a constant speed of $1.4\frac{m}{s}$. Bidirectional communication is considered established between users and edge servers, and among edge servers, if physical distance is within WiFi range ($100m$). Users do not directly communicate with each other. In our experiment, we let the system run for 10 simulated minutes, then we simulate a disruptive event: elected leaders fail with probability ρ—e.g. as would happen due to a city-wise power shortage. After this event, we simulate 10 further minutes of system evolution.

We compare two implementations of the SCR pattern, a classic one (as described in Section 10.2.6) and a version with a feedback loop. In the latter, leaders try to coordinate and resize their regions in the attempt to cover approximately the same number of users, to reduce disparities in elaboration load that would cause slowdowns on overloaded edge servers. We implement self-organising adaptation of region size by feeding the information on the number of served users back to the leader, and using it to dynamically change the region size (the more users, the smaller the region), competing with other leaders. In order to prevent sharp oscillations of the region sizes, with possible resonance phenomena, we don't feed the

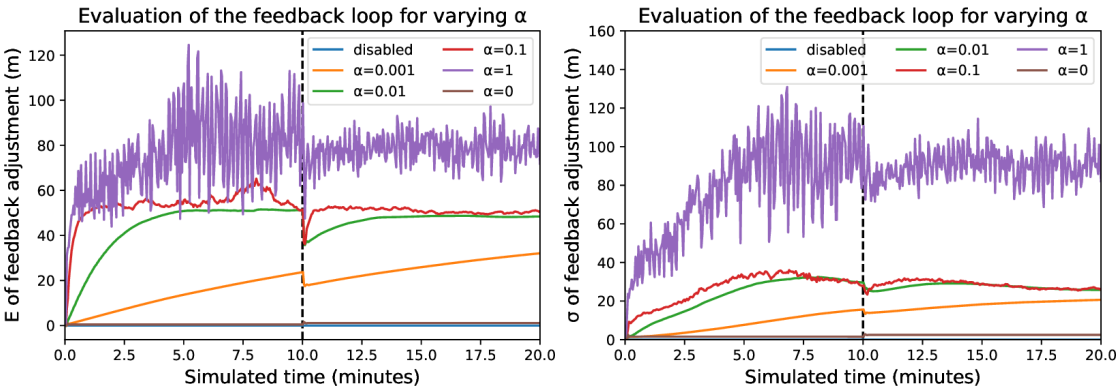


Figure 10.5: Evaluation of the backoff parameter. Values are averaged along all values of u and ρ. Not considering new values ($\alpha = 0$) has a similar effect to disabling feedback entirely. Plugging the feedback directly, without filtering, makes the system oscillate. Other values show how α tunes the trade-off between reactivity and stability, with $\alpha = 0.01$ both smooth and with an impact on the system comparable to $\alpha = 0.1$.

served user count back to the algorithm input directly, but we filter it using an exponential backoff (a low pass filter), namely, the feedback value is $\alpha u_t + (1-\alpha)u_{t-1}$, where u_t is the count of served users at time t.

We first evaluate good values for α in our scenario, by looking at how different values affect the size of regions and their stability. We then measure performance and resilience for both the base and the optimal-α versions of the SCR pattern varying the number of users and ρ, and observe the number of users served in total and by each edge server. A summary of the free variables for the case study is given in Table 10.2; measures are instead summarised and explained in Table 10.3.

The pattern has been implemented in Protelis [PVB15], and simulations have been performed using Alchemist [PMV13]. We executed 100 replicas of the experiment for each configuration in the cartesian product of the parameters values, varying displacement of edge devices, initial position of users and their waypoints, and execution times of devices. Data has been processed using Python xarray [HH17] and matplotlib [Hun07]. The experiments include a reference implementation of the SCR pattern, they are entirely open-sourced, automated, and reproducible using the instructions provided in a publicly accessible repository[4].

[4] `https://bitbucket.org/danysk/experiment-2019-coordination-dynamic-orchestration`.

Results We initially measure the benefits of using the feedback system and the impact of different values for α. Results are depicted and described in Figure 10.5, and show how $\alpha = 10^{-2}$ is the best choice among the analysed values.

We then evaluate correctness and performance of the algorithm both without and with feedback enabled ($\alpha = 10^{-2}$). Results presented in Figure 10.6 show that the system is able to serve all the users, actually serving some users twice at the moment they cross the boundary between neighbouring regions.

Finally, we study resilience of the system to failures by analysing its behaviour with different sudden disruptions hitting the leaders. Figure 10.7 shows the pattern reaches stability in few seconds even when disruption is large, and regardless of the feedback system. At disruption time, several nodes are not served and several others get instead apparently overserved, as they are in an inconsistent state and participating in multiple, quickly changing regions, with their streams getting lost because of the time required to recover both regions and spanning trees for data accumulation. The feedback system has a negligible impact on resilience, but improves load balancing both before and after disruption.

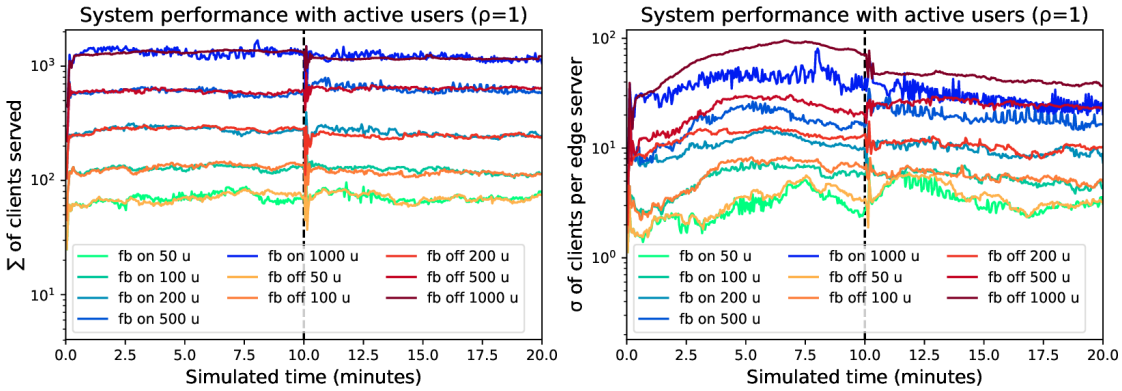


Figure 10.6: System correctness. Warm colours are results with feedback system disabled, cold colours are results with feedback system enabled and $\alpha = 10^{-2}$. Both configurations serve all the users, and actually slightly "overserve" them. This is due to the fact that users joining a different region, have, for some time, their streams counted also in the region they left due to network propagation and elaboration times. The feedback system provides benefits in terms of load balancing, as depicted in the right chart: the lower σ means lower disparity among leaders in the number of served users.

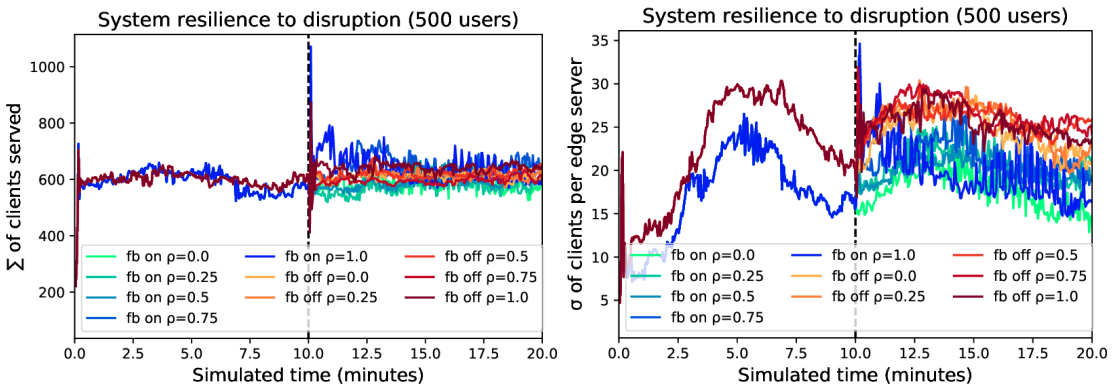


Figure 10.7: System resilience to disruption. Both the pattern configurations provide resilience to disruptions. The system is able to find new leaders in few seconds even if the whole set of previously selected leaders is shut off. The feedbacked system seems to achieve slightly better performance for smaller disruptions, but takes more time to stabilise in the worst case. As seen in Figure 10.6, the feedbacked system achieves visible better performance in terms of load balancing, both before and after the disruptive event, regardless of its entity.

10.3.2 Case study #2: situated problem solving

Situated problem solving system model Our goal is to build a distributed coordination system for large-scale, situated, collaborative problem detection and problem solving. The generalised model we consider is as follows. A system is situated within an *environment*, where *problems* (or *issues*) arise. The environment is inhabited by a (large) set of heterogeneous *agents* (a.k.a. *workers*) making up the IoT system, which roam inside it and interact opportunistically. Workers have *sensors* and *actuators*, to perceive the environment for potential issues and perform repairing actions, as well as specific *skills* (a.k.a. *capabilities*)—i.e., an ontology of pragmatic or epistemic actions potentially useful for the considered problems.

As a running example of a hybrid problem-solving IoT system, consider a wide smart manufacturing facility populated with machinery, mobile robots and humans. Due to the facility's operation, toxic waste may be spilled in unknown places within the floor. Sensors—or roaming human workers—may detect waste, which due to health hazards must be cleaned by specialised robots. Cleaning robots move to the toxic waste spill area and clean it, upon instruction of various edge-level system control entities responsible for decision-making. Since the system

goal is critical—toxic waste is dangerous—the system must be resilient in fulfilling its goal and failure of components must not lead to violation of the system goal. Since the toxic spillage *problem* typically emerges in unknown places, it is to be tackled dynamically by cooperation between different entities (i.e., detected through specialised sensors or human workers, and solved by dispatching cleaning robots) while overall control and coordination must take place in a way that is resilient to failure (e.g., faults in single devices or in the control infrastructure must not lead to global failure).

Therefore, the key entities we consider for collaborative problem solving are:

- *Environment.* An IoT system's spatial operational environment that needs to be monitored—e.g., the physical area of the smart manufacturing facility.
- *Problems.* Within the system's environment, *problems* (or *issues*) may arise that need to be solved. Those are *situated* (i.e., localised in space and time); e.g., a toxic waste spill occurs in a specific area within the manufacturing facility at some specific time point.
- *Workers.* These are active, situated agents (e.g., IoT devices, humans, robots) that inhabit the environment and are part of the system. They which opportunistically wander or profitably visit a set of loci of interest to perform tasks. Workers may be heterogeneous, exposing different capabilities w.r.t. detecting or solving problems. For the toxic waste scenario, things or humans enjoy problem detection capabilities (i.e., sensing toxins), while specialised robots are responsible for solving problems (i.e., actuating—cleaning toxic waste). When a worker detects a problem, it cannot autonomously decide how to deal with it, and must report the issue to another entity responsible for decision-making.
- *Coordinators.* Resource-rich computational entities, deployed on the edge, are responsible for coordinating workers. A coordinator takes decisions about issues it has been notified about, which result in assignment of *tasks* to workers under its supervision. Workers which detect problems notify their coordinator, who subsequently allocates appropriate tasks to worker(s) with the necessary capabilities. Coordinators themselves may differ in their decision-making ability or computational power.

The system entities above are not static: they interact towards the global system goal. When a worker detects a problem, either the worker is allowed to directly handle it, or not. The former case is of course rather simplistic and assumes no need for coordination for solving problems; the worker may solve the problem by itself if it owns needed capabilities, or may delegate tasks to other workers. However, in the latter case, which is the one we focus here, the worker cannot autonomously take decisions about how to deal with the problem, and must report the problem to another entity responsible for decision-making, the *coordinator*. Coordinators are responsible for determining the *assignment* (a.k.a. *allocation*) of problems/tasks to workers. The level of sophistication of such decision-making carried out by the coordinator is of course left to the system designer to implement in a domain-specific manner. The coordinator might elaborate a (partial) *plan*, hence assigning (partial) sequences of actions to workers, or it might just *delegate* the issue to the workers, assuming they have the knowledge to do the planning themselves—in this chapter, we mainly consider the latter option. Additionally, a coordinator is expected to play a role throughout the problem solving process, i.e., by also supervising or monitoring the activity of workers and providing any needed help. Accordingly, workers engaged in a task can provide *feedback* to the coordinator in order to report progress, request further resources, or provide any information useful for the specific and overall workflow.

A smart choice of coordinators is generally desired, where "smart" depends on various factors; typically, this means choosing nodes that both guarantee *(i)* good and uniform spatial coverage of the environment, and *(ii)* balanced coverage of workers—which might be unevenly distributed across space. We support *multiple coordinators*, each of which is responsible for a certain portion of the environment (called an *area*). Decision-making is decentralised, as coordinators control sets of workers independently.

However, components in a system might fail, as is especially the case in the highly distributed, volatile IoT-based systems we target. Assuming workers are generally available, failure of coordinators must be tackled, as workers cannot solve problems by themselves and their coordination is critical for achieving the system goal. To this end, we support dynamic selection of coordinators in case of failure: *candidate coordinators* are system components which (in varying degrees) are able

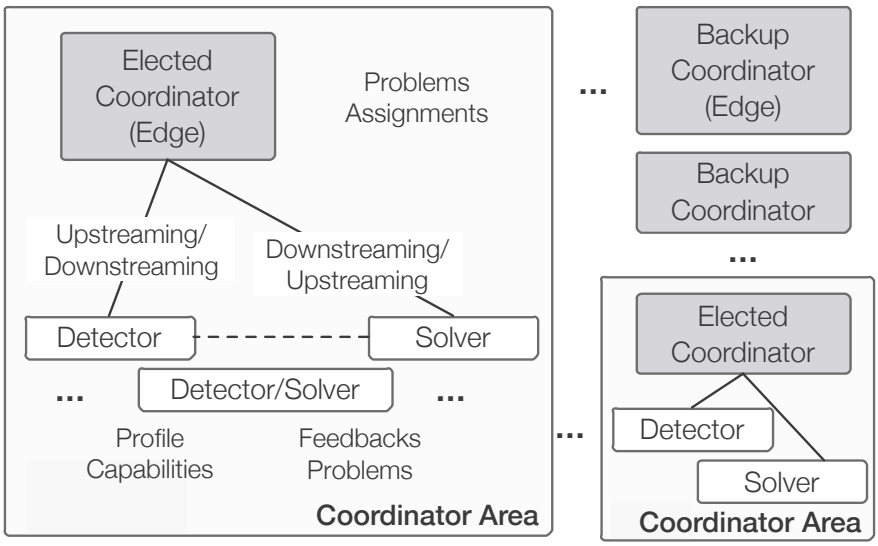


Figure 10.8: Problem-solving ecosystem.

to perform coordination duties (e.g., as being resourceful or trusted enough). Out of candidate coordinators, some *leaders* are elected (active coordinators), responsible for a set of workers; those not elected (i.e., inactive) are considered "backup coordinators". Failure of an elected, active coordinator leads to dynamic, automatic selection of a backup one; thus, the system is resilient to their failure.

We implement the situated, collaborative problem solving workflow of Figure 10.8 through the specification of Figure 10.9.

Problem-solving system concerns The model of Section 10.3.2 defines key abstractions, and relationships among them, essential to the problem solving conception. The system designer can take the problem solving workflow as a functional black box: she just needs to provide inputs/configuration and refine the abstractions with domain-specific details. Methodologically, the following needs to be defined.

- *Problem model* — A taxonomy of the problems has to be defined, together with associated properties and metadata for use, e.g., in allocation decision-making. In the toxic waste removal scenario, a spillage problem can be modelled, e.g., by specifying the location in the environment, the kind of substance, and the rough amount of material to be disposed.

```
class ProblemSolvingEcosystem extends AggregateProgram with ProblemAPI {
  override def main = {
    val coordinators = priorityS(grain, priorityField)
    val potential = branch(infoPropagationNet){gradient(coordinators)}{+∞}
    val problems  = collectSets(downTo=potential, problemOccurrences)
    val solvers   = collectSets(downTo=potential, solverProfile)
    val feedbacks = collectSets(downTo=potential, feedbackField).groupBy(_.
    problem)
    val assignments = branch(coordinators){
      allocate(coordinators,solvers,problems,feedbacks) }{ Set() }
    val tasks       = broadcast(potential, assignments)
    branch(workers){ execute(tasks) }{ () }
} }
```

Figure 10.9: Excerpt of the aggregate program modelling situated problem solving as a decentralised workflow. Gray, underlined symbols denote fields of parameters (e.g., `grain`) or built-in/sensor values (e.g., `solverProfile`). Black, bold symbols denote application-specific functionality. Red and purple symbols denote core and library constructs, respectively.

- *Agents model* — The agents as well form a taxonomy. In the toxic waste removal scenario, we may have human or robot detectors, and three (possibly overlapping) solver roles: waste collector, disposer, and cleaner. So, sensors and actuators have to be defined and provided: e.g., the waste collector may be equipped with a camera, a pump, and mechanical arm. Agents have capabilities—crucial for problem allocation; e.g., waste collectors and disposers may advertise their ability to carry on light or heavy loads, or their resistance to hot or acid substances. Finally, we only assume that an agent is able to communicate , at the minimum, with other nearby agents—the concrete modality being a design decision.
- *Solving processes* — The allocation strategy used by the coordinator to assign tasks/problems to solvers has to be designed, weighing various variables (e.g., required, preferred, and optional skills, solver-to-problem distance, urgency etc.) in an ad-hoc manner, and possibly leveraging heuristics and ML techniques for optimisation purposes. Also, the solving process for each problem type has to be designed in terms of a micro-level workflow, expressed e.g., via finite-state automata. This includes identifying phases and states of the activity, pre- and post-conditions preventing or enabling

progress, and corresponding feedback messages for the coordinator. E.g., the spillage problem solving workflow can be captured as going through states {*start*, *collected*, *disposed*, *cleaned*, *disposed*&*cleaned*}; through feedback, the coordinator can mobilize cleaners and disposers once the *collected* stage is reached.

- *System and environment design* — From the definition of sensors and actuators follows the model of the environment as perceived by an agent. Moreover, other elements of the environment and overall system can be specified, including number of agents; number of edge servers (candidate coordinators); number and dimension of areas; and infrastructural elements such as wireless access points for communication.

Example scenario: infrastructural maintenance in a smart city To evaluate the proposed approach, we illustrate a case study of urban infrastructural maintenance in smart cities and set up an experimental framework with simulations based on the ScaFi-Alchemist platform [CPV16][5]. Our evaluation's focus is on functional correctness, resilience, and on the actual automatic triggering of adaptivity mechanisms.

As a case study, consider a scenario where autonomous agents (e.g., robots) and human workers are collectively employed for maintaining a city's infrastructure. As parts of the city's common facilities may break or degrade, issues must be quickly identified and dealt with appropriate actions. This entails the notification and resolution of issues by the active agents operating within it. Non-autonomous agents might be useful as well: cameras and diffused plain sensors may provide data to smart software components which are capable of inferring semantics and contributing to the system. The issues arising in the city's facilities are *situated*, i.e., they have an identity and location in space-time. Agents may use electromagnetic sensors, smoke/gas sensors, cameras, or even accept inputs by citizens to detect potential problems. Naturally, agents who identify issues might not be able to solve those by themselves: they may not have required skills or enough resources to deal with the problem, and hence they have to report it to a "control centre".

[5]The source code of simulations as well as instructions for running the experiments and generating the plots are available at the repository `https://github.com/metaphori/engineering-collaborative-edge-iot`.

Note that issues might be dealt with, in principle, in a completely decentralised way: the agent who finds an issue may locally broadcast requests for specialists or resources, without involving any central entity, and the closest matching agents would respond.

Such a problem setting fits our approach particularly well. We advocate keeping the system quite decentralised, by splitting it into areas of space of reasonable size, while also introducing centralisation points (the coordinators) to provide more sophisticated/optimised coordination and decision-making. Coordinators should be placed in strategic/central places of the city, and as they may have to optimise decisions, they should be resourceful machines—e.g., a cloudlet [Sat+09] or an edge computer. We assume security countermeasures are taken for the system to be safe, as well as that potential coordinators outnumber required coordinators (e.g., for redundancy) and have legal ability to carry on their tasks. A smart coordinator might choose to allocate problems to workers based on elements like problem severity, skills of workers, or distance from workers to problems; especially when heterogeneous teams are needed to deal with complex issues, such an allocation decision is not an easy one to be left to a self-organising team of workers.

Experimental setup and simulation framework For our experiments, we employ the aggregate specification of Figure 10.9, enriched with simulation-specific code for parametrisation and data gathering. The experimental setting and simulation scenario (depicted in Figure 10.10) are as follows. A number of devices are supposed to be deployed in the city centre of Vienna: 300 lightweight devices and 10 edge servers. Two devices can communicate if they are within 50 meters range. All these devices, including edge servers, are assumed to run the AC middleware as a service and the aggregate application described by the program in Figure 10.9 on top. They are assumed to "fire" (i.e., to run computation rounds and send corresponding data to neighbours) asynchronously but at similar frequencies.

We run simulations considering either "smart" coordinators (which use an advanced allocation strategy of problems to workers—abstracting from the concrete one) or "naive" coordinators, as indicated by a *smartness* boolean parameter. We measure, along time: *(i)* the total number of problems detected by all workers, *(ii)* the problems streamed to the coordinator but still unhandled, *(iii)* the problems

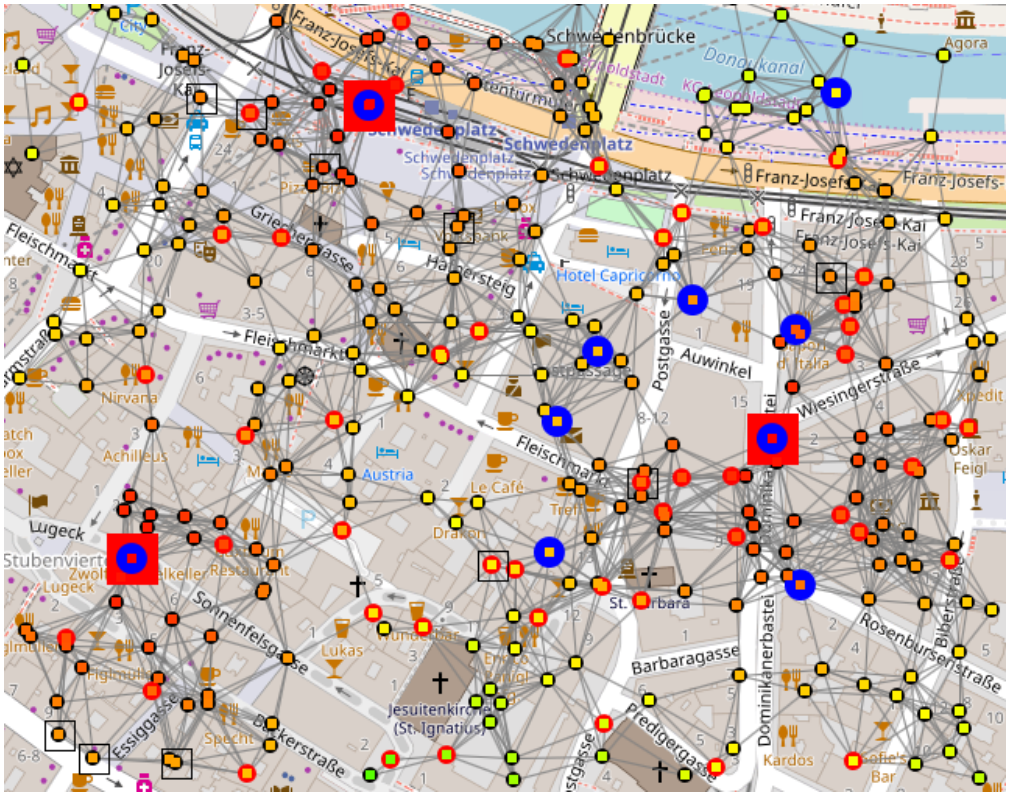


Figure 10.10: Snapshot of the simulation scenario. Large, blue nodes represent edge devices eligible for election as coordinators; large red-filled squares denote leaders. Small circles represent workers; their filling colour reflects the potential field (warmer colours when closer to coordinators). Square contours (e.g., bottom-left corner) denote nodes currently working on a problem. Gray edges depict neighbouring links.

both allocated to *and* accepted by at least one worker (i.e., those successfully assigned), and *(iv)*, the total number of problems handled to completion. We observe the system response by injecting problem occurrences and failure as described in Figure 10.11. Our experiments are implemented as ScaFi simulations [CPV16] and available online.

Experimental results The experimental results are reported in Figure 10.11. Our evaluation goals concern a qualitative assessment of correctness, adaptivity and resilience of the system.

- *Correctness* — Evidence comes from the fact that all the problems found

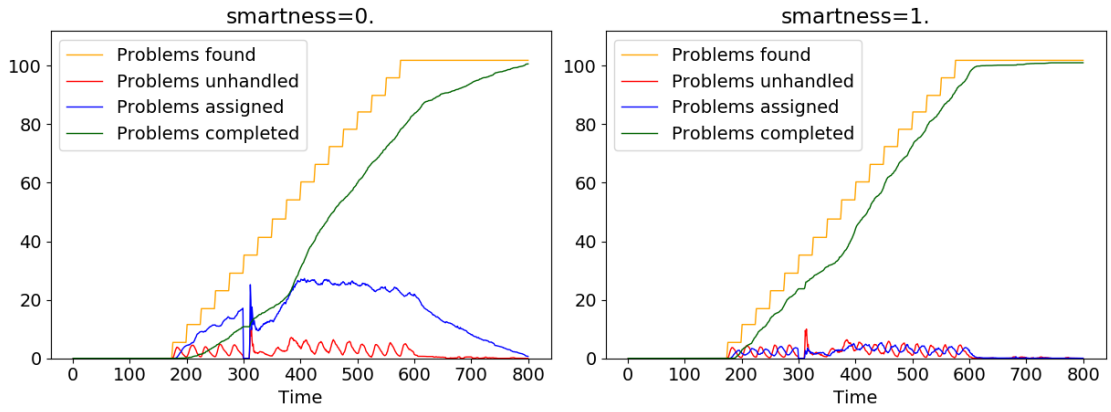


Figure 10.11: Aggregated results of multiple simulation runs, with "naive" (left) and "smart" (right) coordinators. For each case, 50 simulation instances are executed for different random seeds, and the mean values are taken for the measured quantities. From $t_0 = 150$ to $t_1 = 600$ time units, a significant number of "problems" are randomly generated, so that they can be detected by worker devices. Moreover, from $t_2 = 300$ to $t_3 = 310$, a blackout is injected, with the effect of temporarily detaching the edge servers from the network.

have been managed: this means that both the notification of problems, the task allocation, and the feedback process work well. Moreover, notice how the injection of a blackout, disconnecting edge servers from the network (between $t_2 = 300$ to $t_3 = 310$), provides a delay but does not affect the outcome. Finally, differences emerge between coordinators that use an advanced allocation strategy and naive ones: overall, one can observe the increase of performance when smart allocation decisions are taken.

- *Adaptivity* — When the active coordinators fail, the system self-organizes to elect new coordinators. This results into an adaptation of the structures supporting the data flows.
- *Resilience* — Resilience naturally emerges: despite coordination failures, group formation changes or general faults in the control infrastructure, the system does not fail: it correctly responds to failures through appropriate coordination reactions. Notice the gentle degradation of performance caused by failure and the restoration of conventional efficiency.

Discussion

- *Functional Perspective* — Elements about the functional correctness of the solution are empirically verified in this section. The solution schema in Section 7.6.1 is the core of the approach but may not satisfy all the functional properties needed by a real-world application. For instance, the designer needs to decide what happens when an area lacks resources for specific problems or coordinators do not receive timely feedback.
- *Non-Functional Perspective*

  A) *Bandwidth and storage* — The amount of data that needs to be propagated depends on the number of nodes in each area, the amount of problem solving activity, and amount of data required by the coordinators (concerning problem reporting, solver profiles, and feedbacks). Storage is needed because data propagation through aggregate operators `G` and `C` requires keeping state. The specification in Figure 10.9 partially deals with this by using a subset of nodes for the epidemic distribution of data in each area.

  B) *Latency* — In the simplest case, the time between problem identification and resolution is $T_C + T_A + T_G + T_S$ where T_C is the time needed to collect problem data to the coordinator, T_A is the time needed by the coordinator to make its allocation decision, T_G is the time needed to transmit the allocation decision, and T_S is the time needed by the worker to solve the problem. In particular, T_C and T_G are proportional to `grain` and depend on the firing frequency of nodes and the propagation delay, whereas T_A and T_S are application-specific. While it is useful to be aware of these performance aspects, it must be noticed that a variety of optimisations can be applied to the execution process globally sustaining an aggregate application [VCP16]: messages can be compressed (or include only deltas), round frequency can be dynamically adjusted according to desired QoS, communications might be optimised through localisation in edge access points—what is possible ultimately depends on assumptions, configuration, and available infrastructure.

  – *Usability* — A description of what the designer must define and what is provided by our approach is given Section 10.3.2. For the large part of

application design, the designer can focus on the business logic, filling in the gaps with specifications of problems, agents, solution workflows, coordination, and environment. However, knowledge about AC and its toolchain is required for deployment and implementation of extensions with respect to the basic workflow.

- *Generality and Extensions* — The specification of Figure 10.9 represents a general solution schema that can be specialised for different situated problem solving applications, whose key design dimensions are explained in Section 10.3.2. Straightforward examples are those in smart cities, such as infrastructural maintenance (Section 10.3.2), but include generally any scenario involving situated monitoring, decision-making, and action (e.g., firefighting, car crash management, etc.). Depending on the specific scenario, extensions may mix "centralised" and "localised" decision-making, mix "opportunistic" and "planned" monitoring, allow collaboration globally or among adjacent areas, or balance the distribution of skills/resources among areas according to certain metrics (e.g., occurrences of problems, or criticality).

10.3.3 Case study #3: coordinating edge computations

Motivation In this case study, we focus on the problem of *decentralised coordination of edge resources and computations in open scenarios*. The idea is to leverage decentralised coordination, self-organisation, and spatial patterns in order to provide system-level adaptivity and resilience. The system works by dynamically partitioning itself into areas (which can be thought of as edge-clouds) governed by corresponding managers, and setting up downstream and upstream coordination flows from managers to peripheral nodes (i.e., workers and users) and vice versa.

Problem Definition The recurring theme in FMEC is the *smart exploitation* of resources, i.e., the *utilisation of idle resources* from devices that were not usually *fully* considered for computational or storage purposes (e.g., networking and end devices), and the *coordination of resources and tasks* to both extend the possibilities of individual components and attain non-functional advantage in system as well as user processes. In this chapter, we focus on large-scale scenarios char-

acterised by dense groupings of mobile and heterogeneous devices with diverse computational and networking capabilities. The question addressed is: *how can we expose, manage, and coordinate the resources made available by such computational collective in order to build a scalable, adaptive edge computing platform?*

Our reference scenario is a city, more or less smart (in the sense that it might provide no, little, or much infrastructure), that hosts a large number of computational things and agents (possibly mobile, such as people with mobile phones or wearable devices); then, all these devices may *offer/advertise resources* (e.g., à la *volunteer computing* [And10]) or *request resources* to the system—e.g., for task execution. The resource providers and consumers are *situated* entities—i.e., they reside in a environment (which can be perceived and manipulated) and their position is generally relevant since it affects interaction (e.g., who can be contacted, or the cost of communications). Accordingly, we address the problem of building a *system for large-scale, opportunistic, situated resource management and scheduling that spans the thing, edge, and fog layers*. Specifically our goal is to present a design approach that fosters some key properties:

- *Scalability.* The system should be able to scale with the number of devices and the size of geographical deployments (i.e., also with the density of devices). This calls for decentralisation in interaction and decision-making.
- *Minimal connectivity requirements.* Devices do not need to be connected to the Internet; we only assume a device is able to send messages within its neighbourhood.
- *Minimal infrastructural requirements.* The approach should work seamlessly with or without preexisting infrastructure in place, i.e., it could leverage *mobile ad-hoc networking (MANET)* [Bel+13].
- *Opportunism.* The system and its users leverage opportunistic interactions to coordinate and carry out activities.
- *Adaptivity.* The system should be self-adaptive with respect to infrastructural changes as well as perturbations induced by the environment and the autonomous behaviour of agents.
- *Openness.* Components can dynamically enter or exit the system in order to participate in it or not.

- *Resilience and graceful degradation.* Permanent or temporary failures of devices and infrastructural elements should not significantly affect the system functionality.
- *Hybrid resource coordination style.* The system should balance centralised and decentralised decision-making for the allocation of tasks to resources.
- *Global strategies and local tactics.* The system should balance between the exploitation of local opportunities and the pursuing of global-level benefits.

In other words, we make very few assumptions on connectivity and reliability and rather address dynamicity through adaptivity and opportunistic coordination. Also, we trade off performance for adaptivity, as our focus is not on statically computing an optimal resource allocation, but rather supporting edge computations in highly dynamic scenarios.

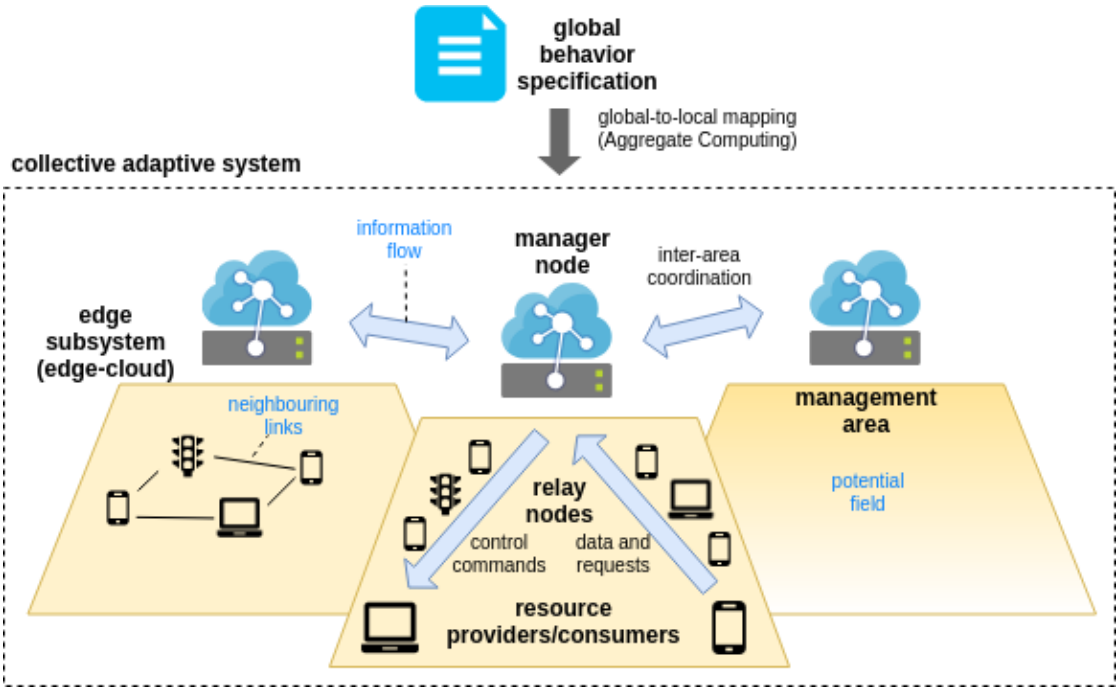


Figure 10.12: Visual overview of the proposed design for self-organising edge-clouds.

Key entities and collaborations We consider an *environment* inhabited by (possibly *mobile*) *devices*. These devices are entities of any sort capable of compu-

tation and networking; they may include user devices (e.g., smartphones or other wearables), IoT devices (e.g., smart light poles, traffic lights), edge devices (e.g., gateways, servers, roadside units) and fog devices (e.g., routers, cellular base stations). We assume the network topology is dynamic and not known apriori: these devices can only interact in an opportunistic fashion by exchanging messages with other devices (also called *neighbours*) located in the vicinity. A device, according to its characteristics and configuration, may play one or more of the following *roles*:

- *Resource providers* (aka *workers*) — These devices offer (a share of their) *resources* to the system and are available for running tasks in a sandboxed environment on the behalf of client devices. They may provide both a share of resources for peer-to-peer negotiation and a share of resources for orchestrated coordination.
- *Resource consumers* (aka *users*, *clients*) — These devices delegate the execution of *tasks* to the system by requesting appropriate resources. Clients may contact nearby workers directly (e.g., within a 3-hop range) or dispatch a request to their master (whose location may be unknown).
- *Manager nodes* (aka *leaders*) — These devices are responsible for managing (i.e., monitoring and controlling) workers and satisfying requests from clients. They are typically resourceful computers (e.g., edge servers or fog nodes), preferably non-mobile, and located in correspondence of hotspots to ensure wide "coverage" of devices.
- *Relay nodes* (aka *links*) — These devices collectively create a mesh network to ensure there exists a hop-by-hop path from workers and clients to orchestrators. The Link role is just an optimisation (with respect to the case in which any device contributes to information spreading) to limit the amount of energy spent in sustaining the system through continuous coordination.

These roles represent a way to support various levels of commitment to the system (for flexibility with respect to local resources and the will to participate) as well as to reason about the different functions that need to be maintained. Notice that all these roles may coexist in a given device.

In order to deal with a huge number of situated devices, we apply the *divide-et-*

impera principle: the environment is partitioned into a number of *(management) areas* (aka *partitions*, *localities*, *regions*, or *edge-clouds*), each managed by a different master. It comes natural to perform such division *spatially*—which is also coherent with the *locality principle*. All the resource providers and consumers refer to the master of the corresponding partition: they upstream data and requests, and receive control data emitted by the master downstream.

We stress that the system should be able to operate in dynamic environments where devices may move or fail; indeed, users, workers, and even managers and relay nodes might be autonomous with respect to many aspects, and only be required to respect the coordination protocol for their role in order to sustain the self-organising behaviour of the system.

The proposed design can be adapted to find suitable trade-offs between centralised and decentralised coordination and decision-making. For instance, simple tasks may be offloaded from clients to workers without requiring any intermediation from the manager; in such case, the manager could just monitor the activity in its area and interact with other managers to perform meta-coordination—e.g., negotiating resources and dispatching reconfiguration of workers among areas.

Design issues and dimensions The presented model is independent of a number of lower-level issues and mechanisms, and is hence highly configurable, e.g.:

- *How are the management areas determined and structured* — Since areas are a notion used to decentralise management activities for scalability purposes, these should be created in order to evenly balance load in the system as well as exploit locality (co-located users may issue similar requests, and co-located workers can interact with low latency). Thus, the management regions should be dynamically created and gracefully adapted by considering the distribution of workers, consumers and managers. Often, managers are determined first, and then regions are negotiated in turn, based on the desired shape of the edge-clouds in terms of exposed resources.
- *How are the managers chosen among eligible devices* — The system should self-organize by finding consensus on leadership among the set of candidates. The point is not the actual algorithm but the properties that need to be enforced and maintained. Generally, the managers should uniformly cover

the situated workers and users, but more advanced choices could also consider the trust, resourcefulness and dependability of manager candidates, or the density and profiles of users and workers.

- *As leaders and areas are dynamically determined, how can workers and users know how to contact the respective manager* — The model abstracts from these details. However, we generally assume every node is at least able to communicate with neighbour devices and there exists a sequence of other devices (i.e., a path of relays) connecting users/workers to leaders. This is a sufficient condition for setting up communication paths (e.g., via patterns like potential/gradient fields [DWH06] unfolding from leaders) to enable self-organising *information flows* [saso07DeW].
- *What should happen when a master node fails* — The failure of a master usually invalidates the invariants required from a configuration of masters. In any case, an area loses its leader and, unless that area could dissolve by feeding adjacent areas, a new leader must be elected. While this happens, requests from users and feedback from workers cannot be handled, but they eventually will be as soon as a newly elected master will receive them.
- *What should happen when a link node fails* — This should not have serious consequences, unless this results in permanent network partitioning. The system should self-organize to properly correct the paths followed by information flows; the aforementioned gradient fields could inherently handle this adaptation.
- *What should happen when a worker node fails* — It essentially depends on what kind of guarantees must be provided by the system. In any case, the leader should become aware of such failure (e.g., by requiring a *heart beat* and considering a temporal threshold that may also depend on a volatility metric characterising the risk for communication delays in the area under supervision). Also, if the assigned task is continuable, intermediate results might be stored in nearby devices or in the master, so that they can be retrieved by a newly allocated worker.

The focus of the approach is on the *coordination* logic for an edge computing platform. However, there are some important concerns (application-specific issues

or challenges on their own from which we abstract from) that should be defined and developed to actually design a working ecosystem:

- *Management functions* — These may include monitoring (e.g., resource usage or availability), control (e.g., assignment of tasks), orchestration (i.e., control of coalitions of workers), or choreography (e.g., managers may group workers into "teams" for cluster computing, and let these work autonomously). Such aspects may affect the logical structure of the system, e.g., whether masters directly communicate with users or always interface with workers.
- *Task scheduling strategies* — Assigning tasks to workers requires to solve issues like task placement and partitioning [Mao+17]. It is reasonable to centralise such issues in manager nodes, which have a global view of the corresponding areas. For optimising choices, the designers should consider what information needs to be collected into those decision points (e.g., QoS preferences and requirements, locations, accurate task models, and so on).
- *Sociality and economics* — The model is configurable with respect to the aspects related to consumption and offering of resources. E.g., in the vision of social clouds [Cha+12], dis/incentives can be used to regulate sharing, trading, and interactions through socially corrective mechanisms.
- *System structure and environment* — Particular applications could impose more or less constraints on the physical and logical structure of the system. Specific decisions should be taken regarding how leaders are elected, how areas are determined, the concrete shape of components, their number and requirements, the assumptions on infrastructure (e.g., WAPs) and so on.

Evaluation The approach is empirically evaluated through synthetic experiments built on the ScaFi-Alchemist simulation framework [CPV16]. In particular, we complete the core implementation schema provided in Figure 10.13 with methods and types to support a service and request management functionality, where (i) workers advertise the services they provide; (ii) consumers can send requests for services to the area manager; and (iii) the area manager handles requests by allocating them to workers or declining them if no worker in the area supports the requested service. The source code of the simulations, launch scripts, and

```
class EdgeCloudCoordinationWorkflow extends AggregateProgram with ... {
  // Some definitions (excluded for brevity)

  def main = {
    // 1) Elect managers among powerful "fog" nodes
    val leaders = branch(FOG){ S(grain) }{ false }

    branchOn(EDGE || FOG) {
      // 2) Build the adaptive communication structure, based on a
      // potential field pointing to leaders for data down-/up-streaming
      val potential = branch(leaders || RELAY){ distanceTo(leaders) }{ +∞ }
      val cs = CommunicationStructure(leaders, potential)

      // 3) Sets up a "continuous" feedback control loop base
      rep(DownstreamData.empty){ case dFlow =>
        val data = branchOn(isWorker || isConsumer){ execute(dFlow) }
        val uFlow = dataUpstream(cs, data)
        val controlData = branchOn(leaders){ processData(uFlow) }
        dataDownstream(cs, controlData)
  } } }

  // Workers/clients receive commands/events from leaders and produce data
  def execute(dd: DownstreamData): Data
  // Data/events by workers/clients are collected/streamed to leaders
  def dataUpstream(cs: CommunicationStructure, data: Data): UpstreamData
  // Leaders process upstream data/events and issue control commands/events
  def processData(ud: UpstreamData): ControlData
  // Control commands/events or area-wide information is sent around areas
  def dataDownstream(cs: CommunicationStructure,
                     cd: ControlData): DownstreamData
}
```

Figure 10.13: Core aggregate implementation schema for an edge computing ecosystem: symbols in bold black and gray are methods and types, resp., to be implemented for application-specific functionality (e.g., using Aggregate building blocks as per [CPV16]); red and purple symbols denote primitive and derived field constructs; blue symbols are Scala keywords.

Property	Solution
Scalability	• Partition into areas covering a subset of components • Decentralised coordination with neighbours
Connectivity/ Infrastructural requirements	• Only local, short-range connectivity is assumed • Independence from the concrete communication technology
Opportunism	• Opportunistic coordination with nearby devices
Adaptivity/ Resilience	• Continuous sensing, interaction, and actuation • Self-organisation
Openness	• Participation only requires executing the program/protocol
Hybrid coordination	• Tunable degree of de/centralisation • Flexibility in decision-making/responsibility distribution

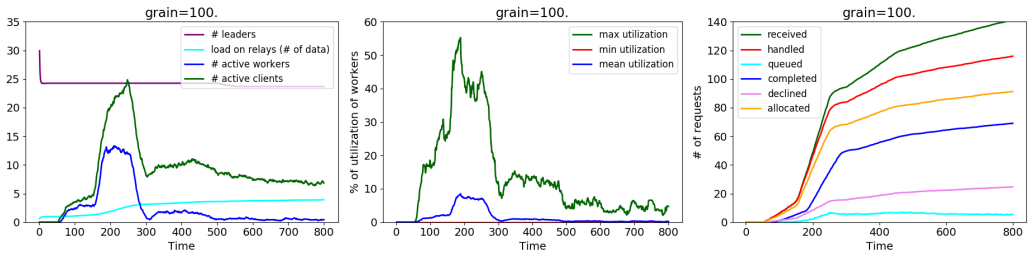
Table 10.4: Characteristics of the edge-cloud coordination solution.

additional information are available at the accompanying repository[6].

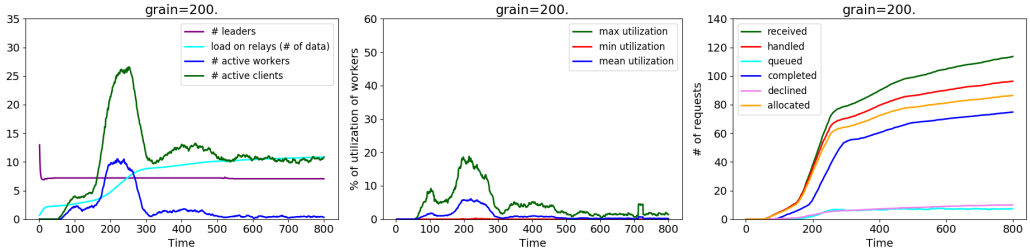
Setup We assume to be in a (smart) city. Our system consists of 50 fog nodes irregularly covering the urban area, 200 worker nodes supporting zero or more services (one of them being rarer than the others) and also working as relays, and 500 clients unevenly distributed in the city. The sleeping period between execution rounds in each device is about 1 second. The fog nodes can interact with one another, whereas clients and workers are connected only with proximate devices (50 metres range). Over time, the clients request services, with a peak in the time frame $[150, 250]$. Multiple simulation configurations are considered, by varying the granularity of the areas. We launch 30 simulation runs with different random seeds for each configuration.

Results and discussion Our goal is to show functional correctness and basic performance. Figure 10.14 shows some key metrics of the system, for different partitioning granularities. With respect to more coarse-grained partitioning, in the case of many, smaller areas (Figure 10.14a), we observe a minor load on relays (as fewer events and less worker/consumer data have to be propagated), but a higher risk of being saturated (i.e., reaching 100% utilisation of worker resources)

[6]https://github.com/metaphori/fmec19-edgecloud



(a) System performance with many, small areas.



(b) System performance with few, large areas.

Figure 10.14: Edge resource coordination: evaluation graphs.

as well as higher reject rate (since fewer workers and a minor variety of resources will be available). By contrast, larger areas can satisfy more requests (since they can generally count on more kinds of resources), and have greater capacity for dealing with localised spikes of activity. However, larger areas also means that higher load is put on managers and relay nodes.

10.4 Final Remarks

In this chapter, we introduce *Self-organising Coordination Regions*, an adaptive coordination pattern for dynamic, opportunistic scenarios where neither complete centralisation nor full decentralisation of control and decision-making are possible or desirable. The pattern fits a problem of potentially growing relevance, and it is particularly suitable for edge systems and for deploying a coordination stance that covers more than pure locality yet without requiring any global coordinator. To show applicability and benefits, we also present three case studies in edge computing, showing that the pattern is able to create semi-independent coordination regions, compute over aggregated information, and propagate results to region members. The pattern is also easily extensible: we show, e.g., how a simple

feedback mechanism could be devised to improve the load balancing across different leaders. We believe the presented pattern, along with easy implementation in ScaFi, can streamline prototyping and development of a wide class of advanced coordination mechanisms, especially in the context of edge computing.

References

[Ale77] Christopher Alexander. *A pattern language: towns, buildings, construction.* OUP, 1977.

[And10] David P Anderson. "Volunteer computing: the ultimate cloud." In: *ACM Crossroads* 16.3 (2010), pp. 7–10.

[Aud+17] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. "Compositional Blocks for Optimal Self-Healing Gradients". In: *Self-Adaptive and Self-Organising Systems (SASO), IEEE International Conference on.* IEEE. 2017.

[Bab+06] Ozalp Babaoglu, Geoffrey Canright, Andreas Deutsch, Gianni A Di Caro, Frederick Ducatelle, Luca Gambardella, Niloy Ganguly, et al. "Design patterns from biology for distributed computing". In: *ACM Transactions on Autonomous and Adaptive Systems* 1.1 (2006), pp. 26–66.

[BE17] Kashif Bilal and Aiman Erbad. "Edge computing for interactive media and video streaming". In: *2nd Int. Conf. on Fog and Mobile Edge Computing (FMEC).* IEEE, 2017.

[Bel+13] Paolo Bellavista, Giuseppe Cardone, Antonio Corradi, and Luca Foschini. "Convergence of MANET and WSN in IoT urban scenarios". In: *IEEE Sensors Journal* 13.10 (2013), pp. 3558–3567.

[Bir07] Ken Birman. "The promise, and limitations, of gossip protocols". In: *ACM SIGOPS Operating Systems Review* 41.5 (2007), p. 8.

[BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. "Aggregate Programming for the Internet of Things". In: *IEEE Computer* 48.9 (2015), pp. 22–30. DOI: 10.1109/MC.2015.261.

[Bus+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley, 1996.

[Cas16] M. Casciaro. *Node.js Design Patterns, 2nd Edition.* Packt, 2016.

[Cas+19a] Roberto Casadei, Giancarlo Fortino, Danilo Pianini, Wilma Russo, Claudio Savaglio, and Mirko Viroli. "A development approach for collective opportunistic Edge-of-Things services". In: *Information Sciences* 498 (2019), pp. 154–169.

[Cas+19b] Roberto Casadei, Christos Tsigkanos, Mirko Viroli, and Schahram Dustdar. "Engineering Resilient Collaborative Edge-Enabled IoT". In: *2019 IEEE International Conference on Services Computing (SCC)*. 2019, pp. 36–45. DOI: 10.1109/SCC.2019.00019.

[CDK05] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. Pearson Education, 2005.

[Cha+12] Kyle Chard, Kris Bubendorfer, Simon Caton, and Omer F Rana. "Social cloud computing: A vision for socially motivated resource sharing". In: *IEEE Transactions on Services Computing* 5.4 (2012), pp. 551–563.

[CPV16] Roberto Casadei, Danilo Pianini, and Mirko Viroli. "Simulating large-scale aggregate MASs with alchemist and scala". In: *Computer Science and Information Systems (FedCSIS), 2016 Federated Conference on*. IEEE. 2016, pp. 1495–1504.

[CV18] Roberto Casadei and Mirko Viroli. "Programming Actor-Based Collective Adaptive Systems". In: *Programming with Actors: State-of-the-Art and Research Perspectives*. Vol. 10789. Lecture Notes in Computer Science. Springer, 2018, pp. 94–122. DOI: 10.1007/978-3-030-00302-9_4.

[CV19] Roberto Casadei and Mirko Viroli. "Coordinating Computation at the Edge: a Decentralized, Self-Organizing, Spatial Approach". In: *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*. 2019, pp. 60–67. DOI: 10.1109/FMEC.2019.8795355.

[Dau+18] Rustem Dautov, Salvatore Distefano, Dario Bruneo, Francesco Longo, Giovanni Merlino, et al. "Metropolitan intelligent surveillance systems for urban areas by harnessing IoT and edge computing paradigms". In: *Software: Practice and Experience* 48.8 (2018), pp. 1475–1492.

[DB16] Soura Dasgupta and Jacob Beal. "A Lyapunov analysis for the robust stability of an adaptive Bellman-Ford algorithm". In: *55th Conf. on Decision & Control (CDC)*. IEEE, 2016.

[DDFM19] Ada Diaconescu, Louisa Jane Di Felice, and Patricia Mellodge. "Multi-Scale Feedbacks for Large-Scale Coordination in Self-Systems". In: *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE. 2019, pp. 137–142.

[DRT05] Manuel Diaz, Bartolomé Rubio, and José M Troya. "A coordination middleware for wireless sensor networks". In: *Systems Communications*. IEEE. 2005, pp. 377–382.

[DWH06] Tom De Wolf and Tom Holvoet. "Design patterns for decentralised coordination in self-organising emergent systems". In: *ESOA'06 Proceedings*. Springer. 2006, pp. 28–49.

[DWH07] Tom De Wolf and Tom Holvoet. "Designing self-organising emergent systems based on information flows and feedback-loops". In: *1st SASO Conf.* IEEE. 2007, pp. 295–298.

[FM+13] Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, Sara Montagna, Mirko Viroli, and Josep Lluis Arcos. "Description and composition of bio-inspired design patterns: a complete overview". In: *Natural Computing* 12.1 (2013), pp. 43–67. ISSN: 1572-9796. DOI: 10.1007/s11047-012-9324-y.

[Han13] Robert S Hanmer. *Patterns for fault tolerant software*. John Wiley & Sons, 2013.

[HCY+99] Sandra Hayden, Christina Carrick, Qiang Yang, et al. "Architectural design patterns for multiagent coordination". In: *Int. Conf. on Agent Systems*. Vol. 99. 1999.

[HH17] S. Hoyer and J. Hamman. "xarray: N-D labeled arrays and datasets in Python". In: *Journal of Open Research Software* 5.1 (2017).

[HL04] Bryan Horling and Victor Lesser. "A survey of multi-agent organizational paradigms". In: *The Knowledge engineering review* 19.4 (2004), pp. 281–316.

[Hun07] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[HW04] G. Hohpe and B.A. Woolf. *Enterprise Integration Patterns*. Prentice Hall, 2004. ISBN: 9780321200686.

[HW08] M. Haklay and P. Weber. "OpenStreetMap: User-Generated Street Maps". In: *IEEE Pervasive Computing* 7.4 (2008), pp. 12–18.

[JDB16] Ward Jaradat, Alan Dearle, and Adam Barker. "Towards an autonomous decentralized orchestration system". In: *Concurrency Computat. Pract. Exper.* 28.11 (2016), pp. 3164–3179.

[JM18] Junchen Jin and Xiaoliang Ma. "Hierarchical multi-agent control of traffic lights based on collective learning". In: *Engineering applications of artificial intelligence* 68 (2018), pp. 236–248.

[KC03] Jeffrey O Kephart and David M Chess. "The vision of autonomic computing". In: *Computer* 1 (2003), pp. 41–50.

[KHA17] R. Kuhn, B. Hanafee, and J. Allen. *Reactive Design Patterns*. Manning, 2017. ISBN: 9781617291807.

[Liu+04] Juan Liu, Jie Liu, James Reich, Patrick Cheung, and Feng Zhao. "Distributed group management in sensor networks: Algorithms and applications to localization and tracking". In: *Telecommunication Systems* 26.2-4 (2004), pp. 235–251.

[LRM08] Rita Lima, Nelson Rosa, and Igor Marques. "TS-Mid: Middleware for wireless sensor networks based on tuple space". In: *22nd AINA Workshops*. IEEE. 2008, pp. 886–891.

[Mao+17] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. "A survey on mobile edge computing: The communication perspective". In: *IEEE Communications Surveys & Tutorials* 19.4 (2017), pp. 2322–2358.

[MC14] Mathieu Magnaudet and Stéphane Chatty. "What should adaptivity mean to interactive software programmers?" In: *Symp. on Eng. Interact. Comput. Sys.* ACM. 2014, pp. 13–22.

[Par09] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. 1st. Pragmatic Bookshelf, 2009. ISBN: 193435645X, 9781934356456.

[Pau+19] Aaron Paulos, Soura Dasgupta, Jacob Beal, Yuanqiu Mo, Khoi Hoang, Lyles J Bryan, Partha Pal, Richard Schantz, Jon Schewe, Ramesh Sitaraman, et al. "A framework for self-adaptive dispersal of computing services". In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE. 2019, pp. 98–103.

[PBV16] Danilo Pianini, Jacob Beal, and Mirko Viroli. "Improving Gossip Dynamics Through Overlapping Replicates". In: *LNCS*. Springer, 2016, pp. 192–207.

[PDV17] Danilo Pianini, Simon Dobson, and Mirko Viroli. "Self-Stabilising Target Counting in Wireless Sensor Networks Using Euler Integration". In: *11th SASO Conference*. IEEE, 2017.

[PMV13] Danilo Pianini, Sara Montagna, and Mirko Viroli. "Chemical-oriented Simulation of Computational Systems with Alchemist". In: *Journal of Simulation* (2013). ISSN: 1747-7778. DOI: 10.1057/jos.2012.27.

[PVB15] Danilo Pianini, Mirko Viroli, and Jacob Beal. "Protelis: practical aggregate programming". In: *Symposium on Applied Computing*. ACM. 2015, pp. 1846–1853. DOI: 10.1145/2695664.2695913.

[saso07DeW] Tom De Wolf and Tom Holvoet. "Designing self-organising emergent systems based on information flows and feedback-loops". In: *1st Conf. on Self-Adaptive and Self-Organizing Systems*. IEEE. 2007, pp. 295–298.

[Sat+09] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. "The case for vm-based cloudlets in mobile computing". In: *IEEE Pervasive Computing* 8.4 (2009).

[SC12] Marco de Sá and Elizabeth F. Churchill. "Mobile Augmented Reality: A Design Perspective". In: *Human Factors in Augmented Reality Environments*. Springer, 2012, pp. 139–164.

[Sch+00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000. ISBN: 978-0-471-60695-6.

[Sto00] S.D. Stoller. "Leader election in asynchronous distributed systems". In: *IEEE Transactions on Computers* 49.3 (Mar. 2000), pp. 283–284.

[VCP16] Mirko Viroli, Roberto Casadei, and Danilo Pianini. "On execution platforms for large-scale aggregate computing". In: *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*. ACM. 2016, pp. 1321–1326.

[Ver15] Vaughn Vernon. *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka*. 1st. Addison-Wesley Professional, 2015. ISBN: 0133846830, 9780133846836.

[Vir+18] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. "Engineering Resilient Collective Adaptive Systems by Self-Stabilisation". In: *ACM Transaction on Modelling and Computer Simulation* 28.2 (2018), 16:1–16:28. ISSN: 1049-3301. DOI: 10.1145/3177774.

[Vir+19] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. "From distributed coordination to field calculus and aggregate computing". In: *Journal of Logical and Algebraic Methods in Programming* (2019), p. 100486. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2019.100486.

[Vli98] John M Vlissides. *Pattern hatching: design patterns applied*. Addison-Wesley Reading, 1998.

[WAC+14] Phillip Walker, Saman Amirpour Amraii, Nilanjan Chakraborty, et al. "Human control of robot swarms with dynamic leaders". In: *Conf. on Int. Robots & Sys.* IEEE. 2014, pp. 1108–1113.

[Wey+13] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, et al. "On patterns for decentralized control in self-adaptive systems". In: *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 76–107.

[YKO03] Osher Yadgar, Sarit Kraus, and Charles L Ortiz. "Hierarchical information combination in large-scale multiagent resource management". In: *Communication in Multiagent Systems*. Springer, 2003, pp. 129–145.

[Zah19] Payam Zahadat. "Self-adaptation and self-healing behaviors via a dynamic distribution process". In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE. 2019, pp. 261–262.

[ZLA10] Chongjie Zhang, Victor Lesser, and Sherief Abdallah. "Self-organization for coordinating decentralized reinforcement learning". In: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems. 2010, pp. 739–746.

Chapter 11

Wrap Up

A l'alta fantasia qui mancò possa;
ma già volgeva il mio disio e 'l velle,
sì come rota ch'igualmente è mossa,

l'amor che move il sole e l'altre stelle.

Dante Alighieri · *Divina Commedia*, Paradiso, Canto XXXIII

In this chapter, conclusions are drawn and perspectives for future work are provided.

11.1 Conclusion

As outlined in Chapter 1, this thesis tackles the general problem of computational collective intelligence engineering, and the specific problem of programming and operating collective adaptive systems. Accordingly, starting from the state of the art in research fields like coordination, multi-agent systems, autonomic computing, collective adaptive systems, and aggregate computing (as reviewed in Part I), contributions (see Part II) have been delivered in terms of (i) a novel, Scala-internal, aggregate programming language (ScaFi, Chapter 7), (ii) an operational abstraction for dynamic collective computations carried out by opportunistic teams of devices (aggregate processes, Chapter 8), (iii) a proof-of-concept

middleware for aggregate systems supporting different architectural styles (ScaFi platform, Chapter 9); and (iv) a decentralised coordination pattern for edge computing (SCR, Chapter 10).

11.1.1 Discussion

ScaFi language With respect to previous field calculus implementations like Protelis (and its predecessor, Proto), ScaFi is an embedded, internal DSL. As a consequence, it has access to all the features of the host language. Among these features, ScaFi inherits the static type-checking of Scala, enabling early interception of bugs at compile-time—providing for a better programming experience with respect to Protelis, which is dynamically typed. Type annotations in ScaFi also foster code readability; additionally, their usage is lightweight thanks to Scala's type inference. Moreover, ScaFi exposes all object-oriented, functional, generic, and modular programming mechanisms, which may be valuable for library designers. However, mixing field constructs with certain host mechanisms (e.g., those affecting control flow) can potentially lead to issues: static and dynamic checks, as well as proper documentation and development awareness, could help to avoid pitfalls. By the point of view of development, ScaFi can leverage Scala tools (parser, compiler, syntax highlighters, linters), whereas Protelis requires (the maintenance of) its own toolchain and proper integration with IDEs. A potential drawback of ScaFi is that ScaFi code can only run on the JVM, whereas, in principle, Protelis code can be mapped to different platforms by implementing a new code generator—ScaFi code could also be mapped, but Protelis has already an infrastructure in place thanks to Xtext. In general, however, by the relief from practical burdens through reuse of tools and functionality from the host language, ScaFi could represent an ideal framework for rapid prototyping of field calculus variants.

Aggregate processes The aggregate process abstraction, implemented as a field calculus extension based on the new `spawn` primitive, does increase the practical expressiveness of the field calculus, as covered in Chapter 8. Essentially, a `spawn` expression represents a generation point for concurrent collective process instances. This is somewhat similar to starting threads in Java or spawning processes in Erlang. Therefore, this mechanism introduces a *concurrency aspect* in aggre-

gate programming which, before, was – though not static – only *a priori* defined (emerging through repeated execution and dynamic evaluation of a pre-defined number of computation branches). The well-known practice of multi-threaded programming could be an inspiration for techniques aimed at the coordination of sets of aggregate processes.

ScaFi platform Writing a field calculus program is a minor part of making up a distributed, aggregate system: a middleware is fundamental to ease the development, deployment, and operation of aggregate applications. The ScaFi platform provides a preliminary actor-based and object-oriented façade API targetting the JVM platform. The actor abstraction, by capturing autonomous entities or flows of control that communicate through asynchronous message passing, could help to reduce the abstraction gap found in the design and implementation of a distributed middleware. However, this is a major engineering challenge, with issues including flexibility (to support multiple, diverse scenarios), versatility (to support multiple, diverse devices), and usability (to reduce programming effort). In particular, complexity arises from the heterogeneity of architectures and infrastructures that may potentially be targetted (as shown in Chapter 6).

Self-organising Coordination Regions pattern Design patterns are key in software engineering, for they capture the knowledge of recurrent problems and associated solutions in specific design contexts. Emerging distributed computing scenarios, such as the IoT, CPS, and Edge Computing, define a novel and still largely unexplored application context, where identifying recurrent patterns can be extremely valuable to mainstream development of language mechanisms, algorithms, architectures and supporting platforms—keeping a balanced trade-off between generality, applicability, and guidance. SCR is a general, decentralised coordination design pattern for partitioned orchestration that aims to provide adaptivity and resilience in large-scale situated systems. This pattern, or variations of it, have been adopted in a variety of contexts, as discussed in Chapter 10. Additionally, the pattern finds straightforward implementation in aggregate computing—making its structure easily recognisable.

11.2 Future Work

Potential future work can be envisioned along the following directions.

Aggregate processes: combinators, API, workflows Chapter 8 covers aggregate functions as well as programming techniques for working with aggregate processes, based on the novel `spawn` primitive. However, more work is needed to come up with a principled, rich API for effectively combining aggregate processes together. In particular, programming (collective) workflows is currently not straightforward. This issue is somewhat related to the aggregate execution model, which is logically continuous and abstracted to the programmers: certain aspects of the dynamics as well as some relationships between static and dynamic behaviour are still not clear and deserve further study.

Aggregate computing middleware: adaptive execution In addition to consolidation, the current ScaFi middleware can be extended for a more sophisticated operational management of aggregate systems. Current support enables *flexible* deployment and execution (design-time, manual). A possible set of incremental extensions could be implemented to make such a support:

- *dynamic* — enabling manual reconfiguration of the system at runtime;
- *adaptive* — enabling automatic reconfiguration at runtime, by reacting , e.g., to available infrastructure; and
- *opportunistic* (or *optimising*) — actively seeking for reconfiguration opportunities in order to improve efficiency or QoS while respecting constraints.

Contracts, deviance, heterogeneity An issue that deserves further investigation relates to what it does actually mean to *program an aggregate*. This is especially interesting when the involved components exhibit (various levels of) *autonomy*. For instance, what happens when individuals do not respect the "aggregate contract"? Moreover, humans and autonomous agents participating into an aggregate may exhibit *deviance*: how can this be dealt with? Also, what does it take to coordinate *heterogeneous* aggregates? Notions of trust (cf. Section 7.6), as used in [JJ18], may prove useful.

Collective intelligence and multi-agent organisational paradigms The ability of aggregate programming of specifying group-wide behaviour by a global perspective could be helpful to structure and institutionalise multi-agent systems. Moreover, aggregate processes do provide means of defining dynamic teams of devices. As a consequence, it would be interesting to consider the integration of collective processes within multi-agent architectures.

Declarative model The aggregate computing paradigm is declarative, and hence delegates to the platform various kinds of aspects related to neighbourhoods and execution. Interesting extensions could be investigated to provide more flexibility and control, e.g., by considering: reactive round execution, reflective control of round execution, and support for multiple `nbr` targets.