Alma Mater Studiorum - Università di Bologna

DOTTORATO DI RICERCA IN

COMPUTER SCIENCE AND ENGINEERING

Ciclo 36

Settore Concorsuale: 09/H1 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Settore Scientifico Disciplinare: ING-INF/05 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

A LANGUAGE-BASED SOFTWARE ENGINEERING APPROACH FOR CYBER-PHYSICAL SWARMS

Presentata da: Gianluca Aguzzi

Coordinatore Dottorato

Ilaria Bartolini

Supervisore

Mirko Viroli

Esame finale anno 2024

Abstract

Italiano

I sistemi IT sono sempre più pervasivi e interconnessi, spinti dall'esplosione dei dispositivi IoT e dagli avanzamenti nel edge-cloud computing, con un impatto crescente sulla società e sull'economia. Una visione moderna di tali sistemi è quella di Cyber-Physical Swarm (CPSW): un grande insieme di dispositivi computazionali distribuiti e profondamente integrati nel mondo fisico che, interagendo localmente con altri simili, esibiscono un comportamento collettivo.

Il lavoro di tesi si concentra quindi sulle sfide d'*ingegneria* uniche per i CPSWs, in particolare nella gestione della complessità derivante dall'*intelligenza collettiva* emergente in questi sistemi. Per affrontare questi problemi, viene introdotto un approccio chiamato language-based incentrato su *aggregate computing*—modello di programmazione top-down che nasce per descrivere comportamenti collettivi su larga scala. Questo paradigma è stato scelto perché facilita la progettazione di comportamenti auto-organizzanti, che sono cruciali per il funzionamento efficiente e resiliente di questi sistemi. L'adozione di un approccio language-based ha portato a progressi significativi sia in metodi ibridi–che combinano soluzioni dichiarative e sub-simboliche–sia in approcci ingegneristici standard per i CPSWs. In particolare, sono stati identificati due principali aspetti su cui concentrarsi sugli aspetti ingegneristici: *pattern di progettazione*–ovvero le soluzioni riutilizzabili ai problemi comuni in un determinato contesto–e gli *aspetti della piattaforma*–cioè l'infrastruttura sottostante che supporta il software aggregato. Per quanto riguarda il primo, sono stati sviluppato nuovi algoritmi, API e metodologie di progettazione. Per il secondo, si è agito a livello di distribuzione ed esecuzione collettiva. Parallelamente, è stato esplorato come l'apprendimento possa essere integrato nel paradigma di riferimento attraverso una roadmap. Quindi, è stato efficacemente combinato l'apprendimento in aggregate computing sia a livello di pattern, con lavori basati sull'apprendimento rinforzato multi-agente, specificamente con la progettazione di programmi collettivi e con un nuovo algoritmo chiamato *field-informed reinforcement learning*, sia a livello di piattaforma con un nuovo approccio che mira a creare schedulers distribuiti per il calcolo collettivo.

Per corroborare i risultati e fornire supporto allo sviluppo, è stato anche progettato uno strumento per la gestione della programmazione di applicazioni ibride chiamato ScarLib.

In definitiva, l'obiettivo della ricerca è fornire un approccio olistico per l'ingegnerizzazione e il deployment efficace di CPSWs su larga scala, adattabili e robusti.

Parole chiave – *cyber-physical swarms, programmazione aggregata, auto-organizzazione, ingegneria del software language-based, intelligenza collettiva, intellingenza di sciame, apprendimento automatico, apprendimento per rinforzo multi agent.*

English

IT systems are becoming increasingly ubiquitous and interconnected, driven by the rise of Internet of Things (IoT) devices and advancements in edge-cloud computing. This evolution is having a growing impact on society and the economy. An advanced perspective on these systems identifies them as Cyber-Physical Swarm (CPSW), which consist of large networks of interconnected distributed computational devices deeply embedded in the physical world, exhibiting collective behaviors.

This thesis explores the unique *engineering* challenges associated with these systems, focusing particularly on managing the complexities arising from their collective intelligence and large-scale nature. To address these challenges, this work introduces a language-based approach centered on *aggregate computing*, which is a top-down programming paradigm designed to describe large-scale collective behaviors. This paradigm was chosen because it facilitates the design of self-organizing behaviors, which are crucial for the efficient and resilient operation of CPSWs. Adopting a language-based approach has led to significant advancements in both hybrid methods–combining declarative and sub-symbolic solutions–and standard engineering approaches. Specifically, we identified two major facets to focus on engineering aspects: *design patterns*–i.e., the reusable solutions to common problems within a given context–and the *platform aspects*–i.e., the underlying infrastructure that supports the aggregate software. Regarding the former, we have developed new algorithms, APIs, and design methodologies. For the latter, we have taken action at the deployment and collective execution levels. Concurrently, we have explored how learning can be integrated into our paradigm of reference through a roadmap. We have effectively combined machine learning with aggregate computing both at the pattern level, with work based on many-agent reinforcement learning, specifically collective program sketching and a novel algorithm called *field-informed reinforcement learning*, and at the platform level with an innovative approach that aims to create distributed schedulers for collective computation. To corroborate the findings and provide development support, we have also designed a tool for managing the programming of hybrid applications called ScarLib.

Ultimately, the goal of this research is to provide a holistic approach for the effective design and deployment of large-scale, adaptable, and robust CPSWs.

Keywords – *cyber-physical swarms, aggregate programming, language-based software engineering, self-organization, collective intelligence, swarm intelligence, machine learning, many-agent reinforcement learning.*

For those who find harmony in complexity.

Acknowledgements

I always told myself that I would write these acknowledgements only at the end of my academic journey. However, it seems I may never want to leave this world, so the time has come to thank all the dear people who have been with me throughout this long journey. I want to emphasize that whatever I am, have done, and will do, is all thanks to you: I have merely done what needed to be done, and you have been my guide. I am endlessly grateful to you all.

First and foremost, I must thank my family for always standing by me, even when it seemed like I was lost. In particular, I want to thank my mom for teaching me what it means to love, my dad for showing me the meaning of dedication and sacrifice, and my grandparents—grandfather Giuseppe for his kindness and grandmother Rosa for her boundless dedication to work, for the meals prepared, and for her zest for life. A special thanks go to my older brother Cristiano; one of the most brilliant people I know. He always encouraged me to do my best and to never give up, he has been a shining star in my darker moments—you are the best, keep it up!

I would also like to thank my lifelong friends, Giovanni, Cri, and Sonia. They have seen so many versions of me and still accept me–I do not know how they manage! You have been by my side in tough times, providing lightness, and for that, I sincerely thank you. I hope you continue to tolerate my craziness. A special thanks go to Mone and Pedro who made me feel at home and believed in me during my university years–you are like brothers to me! A note of gratitude also goes to Jo, my roommate for these last few months, for helping me through difficult times–I apologize for all the therapy sessions you have had to endure! I would like to express my sincere gratitude to my colleagues in Area 4.0. Our exhilarating launches and absurd discussions have made difficult times easier to navigate. I assure you that I will continue to make coffee, even when I sometimes lose my composure. Last but not least, I want to thank Marta–my muse and my reference point for so many years. Much of who I am today is thanks to her. I am still not as radiant as I would like to be, but some of the light I do have comes from the years spent with her. Thank you for everything. A special thanks to my second family, the Luffarelli–thank you for loving me as a son. You will forever be

in my heart.

Turning to the scientific side, I want to express my boundless gratitude to my three main mentors: proff. Mirko Viroli, Roberto Casadei, and Danilo Pianini. To you, I want to dedicate this quote that I read while writing this thesis: "having access to people smarter than yourself is a blessing." Mirko exemplifies the kind of professor people aspire to be–his dedication, love for his work, and ongoing enthusiasm have always pushed me to give my best, because "there are no alternatives to excellence". Roberto has been a reference for my work since my bachelor's thesis, he is for me like a lighthouse guiding me through stormy seas–his advice, work ethic, and endless abilities are something I have always aspired to. A special thank you to Danilo, who has given me so much not just technically and scientifically but also encouraged me to be a better version of myself (in his atypical methods). Additionally, the cohesive group of researchers in Cesena exists primarily due to his unconventional and continuous efforts (e.g., the mandatory Friday Sthop), therefore I must extend another heartfelt thank you to him for that. Special mentions go to Alessandro Ricci for his dedication to teaching and his endless curiosity, and to Giovanni Ciatto, who despite his outward cynicism, shows rare love for his work and displays a humanity hard to find in academia. My only regret is that he beats me badly at Mario Kart. I would also like to thank my students, especially Davide, Angelo, Francesco, and Giacomo. You have taught me much more than I have imparted to you–I hope I've shared some of my joy for this work and given you some valuable guidance. A special note also goes to Nico, you have helped me a lot in this last period of my PhD. Conversations with you consistently enrich my perspective and inspire positive change in my work and thinking! I hope to live up to being a good example for you, just as my mentors have been for me. Give more value to the sentence: Conversations with you always leave me with something good

Further, thanks go to my first office colleagues: dott. Angelo Croatti and prof. Sara Montagna, who were my go-to people in the initial dark months related to COVID of my PhD at our PS Lab. Thank you to prof. Lukas Esterle for hosting me in Aarhus and guiding me attentively. Moreover, during my time abroad, I felt at home thanks to all my dormitory colleagues. You made those three beautiful months better. I would also like to extend my gratitude to the researchers with whom I have engaged in thoughtful discussions, including Stefano Mariani, Mirco Musolesi, Guido Salvaneschi, Andrea Omicini, Giorgio Audrito, Giovanni Delnevo, and Morten From Elvebakken. To those not explicitly mentioned, your time and insights have been invaluable in my growth.

In closing, many friends with whom I have shared memorable moments have not been explicitly acknowledged. To you–from my hometown, to those who were with me in the early stages of my academic journey, and to those who have stood

by me even in the final tough months of my doctoral studies–each of you has left an indelible imprint on my life. I earnestly hope that I have been able to contribute something meaningful to your lives, just as you have enriched mine.

Contents

Abstract **ii**

1 Introduction **1**
1.1 Research Background and Context 1
1.2 Overview and Contribution 2
1.3 Thesis structure 6

Reference **8**

I Background **10**

2 Cyber-Physical Swarms **11**
2.1 Overview 11
2.2 Vision examples 13
2.2.1 Wild-fire monitoring in extensive forests 13
2.2.2 Crowd steering 14
2.2.3 Autonomous vehicles 15
2.3 Characteristics 16
2.4 Related concepts 19
2.5 Final Remarks 23

3 Macro-programming **24**
3.1 The Essence of Macroprogramming 25
3.2 Conceptual Framework 26
3.2.1 Preliminaries 26
3.2.2 Macroprogramming: Definition and Basic Concepts 26
3.2.3 Historical Evolution and Context 27
3.3 Aggregate computing 28
3.3.1 System Model 29
3.3.2 Field-based Programming Model 31

3.3.3 Tools 40
3.4 Final Remarks 47

4 Reinforcement Learning **48**
4.1 Single-agent 50
4.1.1 Markov Decision Process 51
4.1.2 Find a policy given an MDP 52
4.1.3 Find a policy without an MDP 55
4.1.4 Policy Gradient Methods 57
4.1.5 Approximate Solutions 59
4.1.6 Wrap up 60
4.2 Multi-agent 61
4.2.1 Stochastic games 62
4.2.2 Taxonomies 64
4.2.3 Solutions for MARL 67
4.2.4 Wrap up 71
4.3 Many-agent 71
4.3.1 Formalization 71
4.3.2 Solutions for many-agent reinforcement learning (ManyRL) . 72
4.4 Final Remarks 76

References **77**

II Engineering Cyber-Physical Swarms **87**

5 Patterns: Sensing-driven Clustering in Swarms **88**
5.1 Field-based Concurrent Processes 90
5.2 Resilient Dynamic Cluster Formation 92
5.3 Sensing-Driven Clustering 94
5.3.1 Assumptions 94
5.3.2 Problem Definition 95
5.3.3 Adaptive Centroid-based Clustering on Numeric Values . . . 97
5.3.4 Adaptive Clustering Meta-Algorithm 98
5.4 Evaluation 102
5.4.1 Scenario Description 102
5.4.2 Evaluation Goals 103
5.4.3 Simulation Framework 104
5.4.4 Simulations 106
5.4.5 Results 111
5.4.6 Discussion 113

5.5 Related Work . . . 114
5.5.1 Swarm-based Environment Monitoring . . . 115
5.5.2 Related Clustering Models and Problems . . . 115
5.5.3 Related Work on Sensing-based Clustering . . . 117
5.5.4 Related Approaches and Programming Models . . . 118
5.5.5 Related Field-based Algorithms . . . 118
5.6 Final Remarks . . . 119

6 Patterns: Dynamic Decentralization Domains 121
6.1 Motivation . . . 122
6.2 Decentralized situation recognition and action: a case study . . . 123
6.2.1 Requirements and abstractions . . . 124
6.3 Dynamic Decentralization Domains in Practice . . . 126
6.4 Evaluation . . . 127
6.4.1 Experimental setup . . . 129
6.4.2 Results and discussion . . . 129
6.5 Final Remarks . . . 130

7 Patterns: Coordinated Movements and Decision Making 133
7.1 Motivation . . . 135
7.2 API Design . . . 136
7.2.1 Movement blocks . . . 136
7.2.2 Flocking blocks . . . 138
7.2.3 Leader-based blocks . . . 139
7.2.4 Team formation blocks . . . 139
7.2.5 Pattern formation blocks . . . 140
7.2.6 Swarm Planning blocks . . . 141
7.3 Evaluation . . . 143
7.3.1 Case Study: Find and Rescue . . . 143
7.3.2 Discussion . . . 146
7.4 Related Work . . . 148
7.5 Final Remarks . . . 149

8 Language: Reactive-based collective computations 150
8.1 Motivation . . . 152
8.1.1 Self-organization Engineering Approaches . . . 152
8.1.2 Functional Reactive Programming . . . 153
8.2 FRASP Programming Model . . . 156
8.2.1 System Model and (Reactive) Execution Model . . . 156
8.2.2 Programming Abstractions and Primitives . . . 157
8.2.3 Paradigmatic Examples: Self-Healing Gradient & Channel . 159

8.3 Implementation 162
8.3.1 Goals 162
8.3.2 Architecture 164
8.3.3 Implementation details 164
8.4 Evaluation 165
8.4.1 Goals 165
8.4.2 Experimental Setup 167
8.4.3 Results and Discussion 169
8.5 Final Remarks 171

9 Platform: Deployment of Cyber-Physical Swarms applications 174
9.1 Background 176
9.1.1 Pulverized aggregate computing 176
9.1.2 Multi-tier programming and ScalaLoci 177
9.2 Multi-tier pulverised aggregate computing 179
9.2.1 Pulverized architecture in ScalaLoci 180
9.2.2 Definition of deployment kinds 180
9.2.3 Integration with aggregate programming 183
9.3 Implications 184
9.4 Final remarks 186

References 187

III Learning in Cyber-Physical Swarms 201

10 Research Roadmap for Hybrid aggregate Computing 202
10.1 Roadmap 204
10.1.1 Goals and Means 204
10.1.2 Patterns: learning aggregate computing (AC) algorithms . . 205
10.1.3 Platform: learning execution strategies and adaptations . . . 206
10.1.4 Platform: learning system structures and re-structuring . . . 207
10.2 Opportunities and Challenges 208
10.3 Final Remarks 209

11 Patterns: Collective Program Sketching 211
11.1 Aggregate Programs Improvement through reinforcement learning (RL) 213
11.2 Motivation: Building blocks Refinement 213
11.3 Learning Schema 214
11.4 Reinforcement learning-based gradient block 216

11.5 Evaluation 217
11.5.1 Simulation setup 218
11.5.2 Results and Discussion 219
11.6 Final Remarks 221

12 Patterns: Field-informed Reinforcement Learning 222
12.1 Background and Motivation 224
12.1.1 Graph Neural Networks 224
12.1.2 Problem formalization 226
12.1.3 Motivation 227
12.2 Approach Description 227
12.2.1 Architecture, fields and aggregate dynamics 227
12.2.2 Learning algorithm 228
12.3 Evaluation 229
12.3.1 Scenario 231
12.3.2 Goal 232
12.3.3 Training Phase 233
12.3.4 Test phase 234
12.3.5 Baselines 234
12.3.6 Metrics 235
12.3.7 Discussion and Results 235
12.4 Final Remarks 236

13 Platform: Distributed Schedulers for Collective Computations 239
13.1 Background and Related Work 240
13.2 Aggregate Platform Improvement Through Reinforcement learning . 241
13.2.1 Learning Setting 242
13.2.2 Reinforcement learning to Reduce Energy Consumption . . . 243
13.3 Evaluation 244
13.3.1 Simulation Setup 245
13.3.2 Discussion and Results 247
13.3.3 On practical applicability 251
13.4 Final Remarks 252

14 Platform: Toolkit for Hybrid Aggregate Computing 253
14.1 Software Description 254
14.1.1 Core abstraction 255
14.1.2 ScaFi-Alchemist integration 257
14.1.3 DSL for learning configurations 258
14.1.4 Tool usage 260
14.2 Experiments 260

14.2.1 Description 260
14.2.2 Results 261
14.3 Related work 262
14.3.1 Many Agent simulators: 263
14.3.2 Multi-Agent Deep RL libraries: 263
14.4 Final Remarks 265

References **266**

15 Conclusion **274**
15.1 Discussion 275
15.2 Future works 276

List of Figures

1.1 Graphical overview of the thesis contributions with the reference to the thesis structure. 6

2.1 Overview of Cyber-Physical Swarm (CPSW) and its related concepts 12
2.2 Taxonomy of Cyber-Physical Swarm (CPSW) w.r.t related Systems 20

3.1 Overview of macroprogramming 25
3.2 High-level behaviour of an agent in an aggregate system 30
3.3 Field Calculus abstract syntax extracted from [Vir+18a]. 31
3.4 The aggregate programming stack 35
3.5 Resilient coordination operators 36
3.6 Gradient computation in a sparse and dense network of devices. In Figure 3.6a, the output from executing a gradient program is presented, with the source node positioned in the bottom-left corner. Figure 3.6b displays the execution of a gradient program in a dense network, where the source node is situated in the top-left corner. The intensity of the colour serves as a gauge for proximity, with redder hues indicating closer distances. 37
3.7 High-level architecture of the ScaFi toolkit. 42
3.8 Design of the core of ScaFi (DSL). 42
3.9 An Alchemist simulation example 46

4.1 Overview of the RL framework. 50
4.2 General schema of policy iteration (left) and value iteration (right) 53
4.3 Overview of the RL algorithms. 61
4.4 Overview of the multi-agent reinforcement learning (MARL) framework. 62
4.5 Overview of the MARL taxonomies. 64
4.6 Overview of the MARL learning paradigms. 65

5.1 Examples of the dynamics of multiple concurrent gradient processes. 93

5.2 Graphical representation of temperature field distributions used in the simulations. 107
5.3 Snapshots of simulation executions during the sensing-driven clustering. 108
5.4 Overview of simulation results in different clustering scenarios . . . 111
5.5 In-depth analysis of good clustering results. 112
5.6 Main examples of bad clustering results 120

6.1 Overview of the dynamic decentralization domains approach 125
6.2 Scala implementation of the dynamic decentralization domains API 128
6.3 FloodWatch simulation snapshots 130
6.4 FloodWatch simulation 131
6.5 FloodWatch risk evolution 132

7.1 MacroSwarm: architecture overview. 136
7.2 Overview of swarm behaviours expressible with MacroSwarm. . . 141
7.3 Examples of the supported patterns. From left to right: line formation, v-like formation, and circular formation. 142
7.4 MacroSwarm graphical simulations example 144
7.5 Quantitative plots of the simulated scenario in MacroSwarm. . . 147

8.1 The reactive dataflow graph corresponding to the channel example. 163
8.2 Architecture of Functional Reactive Approach to Self-organization Programming (FRASP). 164
8.3 Design of FRASP Domain Specific Language (DSL). 166
8.4 Evaluation scenarios implemented with FRASP 167
8.5 Simulation results of FRASP simulations 172
8.6 behaviour of the channel in response to changes in a destination . . 173

9.1 Pulverization model and corresponding ScalaLoci specification. . . . 181
9.2 Examples of pulverized architectures. 182

10.1 The aggregate computing stack 203
10.2 Overview of the research roadmap for combined aggregate computing with machine learning 204

11.1 Integration of reinforcement learning (RL) within the AC control architecture for collective program synthesis 212
11.2 Reinforcement Learning schema used in program synthesis simulations 214
11.3 Performance of our RL-based gradient algorithm with `velocity` = 20. 220

12.1 Phenomenon covering overview 224

12.2 High-level description of Field-Informed Reinforcement Learning (FIRL) approach. 228
12.3 Simulations of the case study scenario in Alchemist for FIRL. . . . 229
12.4 Training results of FIRL. 237
12.5 Quantitative test results of FIRL 237
12.6 Coverage of two zones using the different modes of the controller. . 238

13.1 Description of the general scheme of RL applied to the execution platform. 242
13.2 Pareto front varying the w parameter in the scheduling scenario . . 247
13.3 Scheduling dynamics after the learning phase 248
13.4 Simulation results of Q-learning applied to schedule aggregate computations 249
13.5 Shows the average error and the average ticks during the learning episodes of SWAP scenario. 250
13.6 Error and energy saving percentage (see Section 13.3.1) as nodes vary. 250

14.1 ScaRLib main modules 255
14.2 ScaRLib core architecture 256
14.3 Examples of developed System dynamics in ScaRLib 257
14.4 ScaRLib `alchemist-scafi` architecture 258
14.5 cohesion collision reward function 262
14.6 Cohesion and collision experiment results 263
14.7 Snapshots of the learned policy in ScaRLib 264
14.8 The performance of the learned policy in ScaRLib 264

Listings

5.1 ScaFi pseudo-code of the clustering meta-algorithm 101
8.1 Gradient implemented with FRASP 160
8.2 Channel implemented with FRASP 161
11.1 ScaFi-like pseudocode description (implemented in the simulation) for value-based RL algorithm applied AC. `state`, `update`, `reward` are block specific. 215

Chapter 1

Introduction

Contents

1.1 Research Background and Context 1
1.2 Overview and Contribution 2
1.3 Thesis structure 6

1.1 Research Background and Context

Computation is *everywhere*. This statement is not just a catchy phrase but a reflection of our current reality, shaped by the accelerated development of information technology. Indeed, computation has been seamlessly integrated into our everyday lives, so much so that it often goes unnoticed. Its *ubiquitous* presence transcends traditional boundaries, not just in professional settings but also in our homes, transportation systems, and even our bodies, as proved by *wearable* technologies. This phenomenon is most prominent in the explosion of Internet of Things (IoT) devices. According to recent statistics [1], there are currently over 15 billion IoT devices globally, and this number is expected to surpass 30 billion by the year 2030.

The widespread adoption of computational technologies kicks off a transformative phase for the IT landscape, starting from the *edge* of the network where connected devices are becoming more sophisticated and efficient. These edge devices are not merely passive data collectors but are now capable of *localized* computation (thus *cyber-physical*), storage, and analytics. Moving towards the core, *cloud*

[1] https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

infrastructures are also undergoing significant metamorphosis. Traditional centralized data centres are evolving into a more distributed, decentralized, and dynamic architecture to meet the ever-changing demands of users and applications.

In this context, the term *edge-cloud continuum* suitably encapsulates the current IT landscape. It symbolizes a *fluid* environment where the edge and the cloud are not isolated silos but exist as two poles on a spectrum.

This fluid computational landscape is backed by contemporary theories like ubiquitous [Wei99], pervasive [SM03], everyware [Gre10], and collective [Abo16] computing. These paradigms collectively announce an era where computation is not confined to specific devices but emerges as an integrated capability across a vast, interconnected ecosystem in which a vast array of simple devices interact in a decentralized fashion, leveraging their collective might to accomplish intricate tasks often far beyond the capability of individual units. Drawing from natural systems, it is striking how these intricate computational networks mimic the complex collective behaviours seen in nature. For example, certain social insects, like ants and bees, display remarkable examples of resilient, effective, and efficient self-organizing systems. Such natural phenomena serve as powerful metaphors for understanding and conceptualizing the self-organizing properties within computational systems.

These observations have culminated in the formulation of the concept of Cyber-Physical Swarms (CPSWs), an ensemble of *heterogeneous* computational entities deeply integrated with the physical world which, locally interact with each other, producing a collective outcome. This heterogeneous nature enriches the system's adaptability, robustness, and overall performance, thereby enabling it to solve problems in a more nuanced manner. Instances of CPSWs are becoming increasingly prevalent and varied, seen not only in the realm of swarm robotics but also in human crowd dynamics and the field of IoT devices. In each of these cases, the core principles remain consistent: the employment of self-organizing mechanisms to achieve collective goals in an efficient, resilient, and *adaptable* manner.

1.2 Overview and Contribution

Addressing the engineering challenges of CPSW requires innovative approaches, as traditional device-centric and bottom-up methodologies fall short. The complexities involved in these swarms stem from a myriad of factors, including the nuanced interaction between local and global dynamics, the intricacies associated with distributed control systems, the evolving landscape of IT architectures, and significant scalability considerations. Given these multifaceted challenges, this thesis introduces a comprehensive *language-based* approach that incorporates robust models, cutting-edge techniques, and pioneering algorithms. The objec-

tive is to streamline the design and deployment of self-organizing behaviours in CPSWs that are both *predictable* and *adaptable*. This is achieved by integrating established manual design methodologies, namely aggregate computing [BPV15], with cutting-edge advances in machine learning, such as multi-agent reinforcement learning (MARL) [BBD08]. This *hybrid* approach aims to harness the strengths of both paradigms to create a more robust, efficient, and scalable CPSW framework.

The research underpinning this work draws from a wide array of interdisciplinary fields, such as swarm robotics [Bra+13], multi-agent systems [DKJ18], collective adaptive systems [Fer15], field-based coordination [MZ06], and multi-agent reinforcement learning. By synthesizing insights from these diverse domains, the proposed approach aims to offer a holistic solution for the effective engineering and deployment of Cyber-Physical Swarm (CPSW).

Problem statement

The current state-of-the-art solutions in both automatic and manual design are limited in handling the complexities arising from the collective intelligence of CPSW. While the former may offer gold-standard solutions through a "learn-by-doing" approach, they often struggle to generalize across different scenarios. On the other hand, manual solutions may excel in modularity and declarative design but frequently fall short when it comes to managing complex environmental conditions.

Research questions

RQ1: *What is the right model for engineering such applications?*

A model serves as a simplified representation of reality, tailored for a specific purpose. In the realm of CPSW, it is important that the model encapsulates the system's *collective stance*, enabling both its design and deployment. Consequently, this thesis has devoted considerable effort to devising a model that is both *general*—applicable across various systems— and *specific*—capable of reflecting the unique demands of CPSW.

RQ2: *Does a hybrid approach that combines both automatic and manual design offer any advantages?*

The hybrid methodology represents an innovative strategy to overcome the inherent drawbacks of purely automatic or manual designs. This approach seeks to amalgamate the scalability of automatic design with the flexibility of manual design. Given its pioneering nature, this thesis has thoroughly investigated the hybrid approach's benefits and limitations, assessing its feasibility for engineering CPSW.

RQ3: *Are there specific requirements for CPSW that differentiate them from other large-scale distributed systems?*

the unique requirements of CPSW, as opposed to other large-scale distributed systems, primarily stem from its collective aspect. Thus, elucidating CPSW's particular needs is vital for identifying the shortcomings of existing solutions and for advancing new strategies that effectively address these challenges.

RQ4: *How does the engineering of these applications influence the design process for collective controllers?*

The engineering facet of CPSW plays a pivotal role in the creation of collective controllers. In this context, the design process is not just about creating a controller but also about ensuring that it is capable of adapting to the system's collective dynamics. Therefore, understanding the interplay between engineering and design is crucial for creating effective collective controllers. In this thesis, we have addressed these research questions through a combination of theoretical and practical investigations.

Contributions

The thesis contributes to an area called "language-based engineering" (Figure 1.1), where models, algorithms, machine learning solutions, and tools are constructed around a specific programming language within a given context. In the realm of software engineering, a language-based approach is fundamentally concerned with describing solutions using high-level abstractions provided by a specific programming language. One of the unique attributes of this methodology is its *domain-specific* nature. In essence, it starts by establishing a reference framework, constructs the necessary abstractions to articulate the possible behaviours of a given application, and then employs a Domain-Specific Language (DSL) to formulate solutions. For this dissertation, we have selected CPSW as our broad reference class of systems. We observed that the existing paradigm of aggregate computing (AC) is particularly well-suited for describing emergent behaviours straightforwardly and effectively. This suitability arises from the top-down nature of AC and its capacity to express self-organizing emergent behaviours. Building on this foundation, we adopted a layered approach to bridge the gaps between aggregate computing and the broader domain of reference. In this endeavour, we initially sought to identify the *facets* to target in order to close this gap, which can be broadly categorized into *platform*, *language*, and *design patterns*. At the platform level, we considered all aspects related to executing and deploying a program written in the reference language. At the language level, we aimed to determine if

any unique features are particularly relevant to CPSW. These choices could also be influenced by platform-specific factors, such as the optimal way to run a certain program. Finally, above the language layer, we attempted to identify a set of design patterns, encompassing algorithms, APIs, and libraries, that could be effective in describing collective behaviours in CPWS. Then, we have utilized both standard engineering techniques–proposing new programming models specifically tailored for the intricacies of CPWS and identifying patterns that bring programmers closer to this complex landscape–as well as *unconventional* methods. The latter, which we term as "hybrid," is based on the facets identified above and consists in integrating Artificial Intelligence techniques to either enhance certain aspects of the existing solutions (i.e., improve the efficiency of collective computation) or improve the learning process itself (i.e., reduce the learning time of a given problem). Lastly, this research offers a modern perspective on software engineering, made increasingly relevant by the advent of large language models like GPT [FC20] and LLaMa [Tou+23]. The role of such AI models in assisting and guiding the software development process can no longer be ignored, signalling a new era where human programmers are augmented by intelligent systems capable of facilitating more effective development strategies. Specifically, our contributions toward a hybrid approach include:

1. developing a comprehensive roadmap for integrating aggregate computing with machine learning;
2. introducing a collective program synthesis method to establish robust self-organizing behaviours;
3. proposing a distributed scheduling solution to accelerate the convergence of collective structures expressed through aggregate computing;
4. presenting a novel multi-agent reinforcement learning technique, termed "field-informed reinforcement learning", designed to create robust distributed controllers for large-scale computations;
5. creating a tool called ScaRLib, which supports hybrid aggregate computing by combining state-of-the-art deep learning libraries with the aggregate computing toolchain.

Meanwhile, contributions toward standard engineering approaches include:

1. introducing a novel programming language called FRASP, which is designed to facilitate the engineering of self-organizing behaviours in CPSW;
2. developing a set of 'swarm-like' patterns for coordinated movement, distributed sensing and sensing-based clustering;

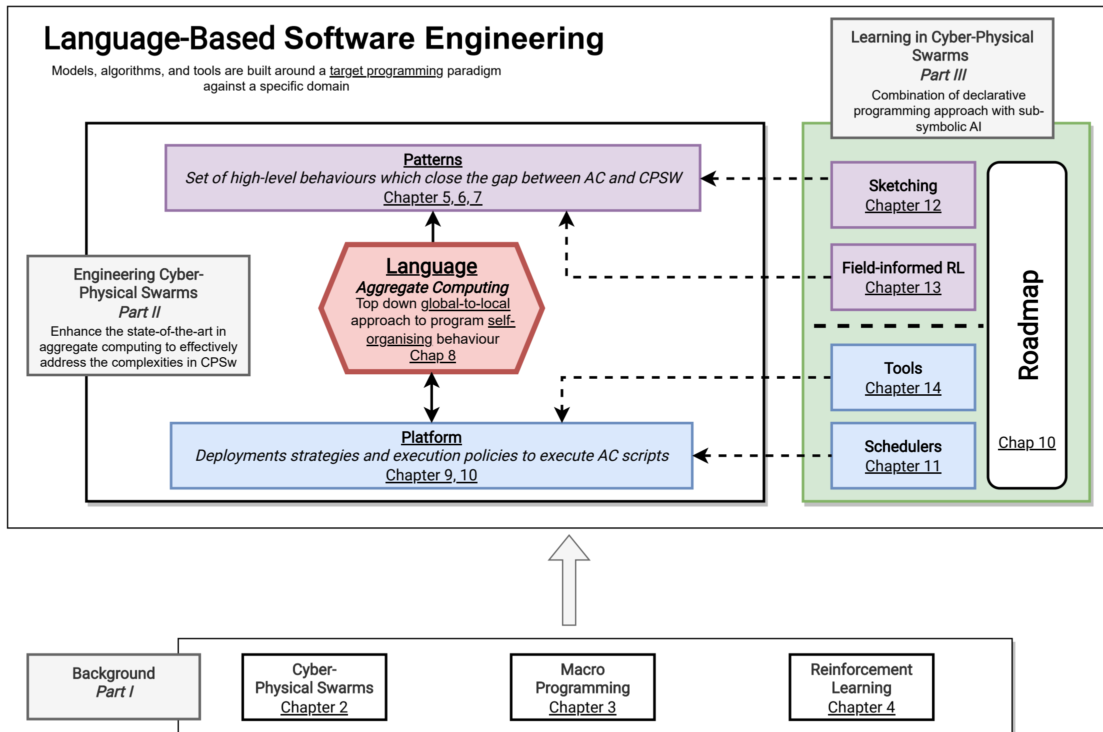


Figure 1.1: Graphical overview of the thesis contributions with the reference to the thesis structure.

3. A novel architecture for deployment on the edge-cloud continuum through a multi-tier pulverized architecture.

1.3 Thesis structure

This thesis is organized as follows. Chapter 1 sets the stage by presenting an exhaustive overview of the research context, elucidating the critical role of language-based engineering in complex systems. This chapter also provides an outline of the thesis, enumerating its key contributions and thematic structure.

Part I lays the groundwork by diving into the theoretical pillars that sustain both standard and hybrid approaches in this domain. It comprises Chapter 2, which articulates the specific characteristics and challenges posed by CPSWs, and Chapter 3, which offers an in-depth explanation of the aggregate computing paradigm and its associated programming model. Finally, Chapter 4 provides an exhaustive overview of the current landscape of reinforcement learning, setting the stage for the terminologies and concepts deployed in subsequent chapters.

Part II focuses on the practical engineering aspects pertaining to CPSWs. It starts with Chapter 5, introducing a groundbreaking algorithm designed for sensing-based clustering that has applications in collective decision-making. Next, Chapter 6 unveils a design pattern adept at encapsulating collective decision-making processes influenced by environmental variables. Chapter 7 elaborates on a pioneering API for enabling coordinated movement and collective choices. Furthermore, Chapter 8 presents an innovative programming model engineered specifically for CPSWs, aiming to enhance the efficiency of collective computations. Finally, Chapter 9 expounds a contemporary deployment strategy that employs aggregate computing across the edge-cloud continuum.

Part III focuses on hybrid learning methodologies applicable to CPSWs. Chapter 10 sketches a comprehensive roadmap for the unification of aggregate computing and machine learning. Chapter 11 introduces a groundbreaking technique for synthesizing collective programs. In addition, Chapter 12 presents an inventive approach to reinforcement learning within CPSWs, grounded in field calculus. Chapter 13 details a cutting-edge strategy for distributed scheduling in these complex systems. Concluding this part, Chapter 14 introduces an advanced toolkit designed for hybrid aggregate computing, leveraging state-of-the-art deep learning libraries.

Finally, Chapter 15 synthesizes the thesis contributions, drawing conclusions and laying out avenues for future research in this evolving field.

Reference

[Abo16] Gregory D. Abowd. "Beyond Weiser: From Ubiquitous to Collective Computing". In: *Computer* 49.1 (2016), pp. 17–23. DOI: 10.1109/MC.2016.22. URL: https://doi.org/10.1109/MC.2016.22.

[BBD08] Lucian Busoniu, Robert Babuska, and Bart De Schutter. "A comprehensive survey of multiagent reinforcement learning". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 38.2 (2008), pp. 156–172.

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

[Bra+13] Manuele Brambilla et al. "Swarm robotics: a review from the swarm engineering perspective". In: *Swarm Intelligence* 7 (2013), pp. 1–41.

[DKJ18] Ali Dorri, Salil S Kanhere, and Raja Jurdak. "Multi-agent systems: A survey". In: *Ieee Access* 6 (2018), pp. 28573–28593.

[FC20] Luciano Floridi and Massimo Chiriatti. "GPT-3: Its nature, scope, limits, and consequences". In: *Minds and Machines* 30 (2020), pp. 681–694.

[Fer15] Alois Ferscha. "Collective adaptive systems". In: *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*. 2015, pp. 893–895.

[Gre10] Adam Greenfield. *Everyware: The dawning age of ubiquitous computing*. New Riders, 2010.

[MZ06] Marco Mamei and Franco Zambonelli. *Field-based coordination for pervasive multiagent systems*. Springer Science & Business Media, 2006.

[SM03] Debashis Saha and Amitava Mukherjee. "Pervasive Computing: A Paradigm for the 21st Century". In: *Computer* 36.3 (2003), pp. 25–31. DOI: `10.1109/MC.2003.1185214`. URL: `https://doi.org/10.1109/MC.2003.1185214`.

[Tou+23] Hugo Touvron et al. "Llama: Open and efficient foundation language models". In: *arXiv preprint arXiv:2302.13971* (2023).

[Wei99] Mark Weiser. "The Computer for the 21st Century". In: *SIGMOBILE Mob. Comput. Commun. Rev.* 3.3 (July 1999), pp. 3–11. ISSN: 1559-1662. DOI: `10.1145/329124.329126`. URL: `https://doi.org/10.1145/329124.329126`.

Part I

Background

Chapter 2

Cyber-Physical Swarms

What is a Cyber-Physical Swarm (CPSW)?
What are the main characteristics of CPSWs?
How do CPSWs differ from other systems?
– **RQ1, RQ3**

Contents

2.1 Overview . . . 11
2.2 Vision examples . . . 13
2.2.1 Wild-fire monitoring in extensive forests . . . 13
2.2.2 Crowd steering . . . 14
2.2.3 Autonomous vehicles . . . 15
2.3 Characteristics . . . 16
2.4 Related concepts . . . 19
2.5 Final Remarks . . . 23

This section aims to elucidate the concept of *Cyber-Physical Swarms*, distinguish systems that align with our conceptual framework (an overview is given in Figure 2.1), and identify research domains grappling with analogous challenges.

2.1 Overview

The term refers to a network of computational nodes working in concert to address collective problems, akin to naturally occurring swarm phenomena —in the

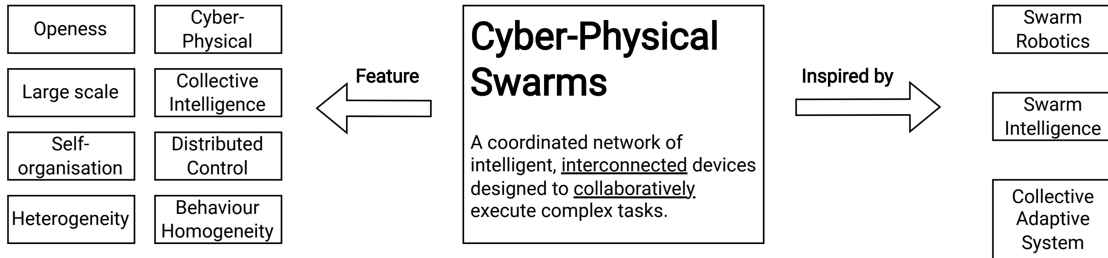


Figure 2.1: Overview of Cyber-Physical Swarm (CPSW) and its related concepts

following will be also referred to as "swarm" systems or "swarm-like" systems. The architecture of these systems comprises a large array of interconnected nodes, numbering in the hundreds or thousands. Importantly, these networks are *open*, implying that the total node count is not predetermined. As a result, the emergent collective behaviour should exhibit *scale-independence*, ensuring that the same programmatic logic applies across small and expansive networks alike. In these systems, each node is tethered to a *physical* entity through sensors and can influence its environment via actuators. This integration of computational and physical elements justifies the term "cyber-physical". The nodes may exhibit heterogeneity in hardware or other attributes, yet they maintain a *homogeneous* local behaviour, executing identical local algorithms. Nodes operate with a focus on collective goals, as opposed to individualistic or selfish objectives, thereby adhering to a collaborative ethos. However, each node can still have local objectives, which are not necessarily aligned with the global ones. Given that centralization is not feasible, the architecture relies on distributed control mechanisms. While modern computing paradigms such as cloud or edge infrastructures are applicable, the system must be capable of swift local responses.

In concluding, it is pertinent to clarify that the focus of this study is on the *macro-level* behaviours of the swarm, rather than the *micro-level* interactions. Rather than characterizing the collective through emergent properties, the intention is to define a *globally* desirable structure.

In the subsequent section, the concept of CPSWs will be elucidated through a range of examples. These will encompass both existing and prospective applications to provide a comprehensive understanding of the field.

2.2 Vision examples

2.2.1 Wild-fire monitoring in extensive forests

Canada is renowned for its lush forests, which cover approximately one-third of the nation's landmass, making it one of the countries with the largest forested areas globally. These green expanses are not only a source of natural beauty but also play a crucial role in maintaining ecological balance. However, the increasing impact of climate change has led to a surge in wildfires, causing widespread devastation to both local flora and fauna. Consider for instance the 2023 wildfires, which burned over 43 million acres, that is about 5% of the entire forest area of Canada [Wik23b].

To address this pressing issue, there is a growing consensus on the need for *proactive* and *localized* interventions. Specialized monitoring systems are essential for keeping tabs on vulnerable regions, especially given the impracticality of relying solely on human observation due to the vastness of these areas. One innovative solution might be the deployment of a sophisticated network of environmental sensors, strategically placed to monitor temperature, humidity, and other fire-prone conditions. These sensors might be complemented by a swarm of drones equipped with advanced imaging and data collection capabilities.

The system operates through ongoing collaboration between ground-based sensors and aerial drones. The drones provide a bird's-eye view of the landscape, allowing for real-time monitoring of a wide array of nodes across large geographical expanses. This integrated approach enables the early detection of potential fire hazards, thereby facilitating the pre-emptive mobilization of emergency services.

However, the implementation of such a comprehensive monitoring system comes with its own set of challenges. Each sensor or drone has a limited operational range and can only provide a *partial* view of the overall system. Therefore, robust data fusion algorithms are necessary to merge information from multiple sources into a cohesive and actionable overview. Environmental conditions are also highly dynamic, requiring the system to *adapt* in real-time to changing variables such as wind speed, temperature fluctuations, and precipitation levels.

In specific regions where the risk is elevated —such as areas with active fires or reduced visibility due to smoke or fog—there may be a need for deploying additional sensors and drones. Given that drones have limited battery life and need to be recharged, the system must be capable of *self-organizing* to ensure uninterrupted monitoring. This is particularly challenging considering that the number of deployed devices could easily exceed thousands of units across the extensive area of interest.

Moreover, the system must be highly responsive to emergency conditions. This necessitates that each device, whether a sensor or a drone, should be equipped with *edge computing* capabilities for performing local analyses. These local analyses can

then be used to trigger collective alarms, ensuring immediate action is taken to mitigate the risk of wildfires.

2.2.2 Crowd steering

Concert venues (or public events like soccer matches) are often filled with an energetic and enthusiastic crowd, eager to enjoy live performances. These events can attract tens of thousands of attendees, making crowd management a critical concern for both safety and enjoyment. However, traditional methods of crowd control, such as barriers and security personnel, are increasingly proving to be inadequate in the face of evolving challenges like sudden surges or emergencies. Take, for example, the 2019 stampede at the San Carlo Place in Turin, where a sudden downpour led to chaotic movements among the crowd, resulting in thousands of minor injuries and three deaths [Wik23a].

To address this complex issue, there is a growing interest in leveraging technology for more effective crowd steering. One promising approach might be to equip each attendee with *smart* bracelets or utilize their smartphones, both of which have computational capabilities. These devices can communicate only with each other, forming a dynamic and adaptive network.

The system functions through real-time data exchange between the individual devices. The individual smart devices provide a “ground-level” perspective, allowing for a granular understanding of crowd behaviour.

This collaborative approach enables the system to identify potential issues before they escalate. For instance, if a particular section of the venue becomes too crowded, the system can send alerts or directions to the smart devices in that area, advising attendees to move to less crowded sections. This facilitates the proactive redistribution of the crowd, thereby averting potential safety hazards.

However, implementing this advanced crowd-steering system is not without challenges. Each smart device only has a *limited* computational capacity and can provide just a *partial* view of the overall crowd dynamics. Therefore, sophisticated algorithms are needed to aggregate this fragmented data into a comprehensive and actionable overview. Additionally, the system must be able to *adapt* rapidly to changing conditions, such as sudden weather changes or unexpected incidents during the concert.

In specific situations where immediate action is required–such as medical emergencies or security threats–there may be a need to send targeted alerts or instructions to specific groups of attendees. Given that smart devices have limited battery life, the system must also *self-organize* to prioritize critical alerts and instructions, ensuring that crowd management remains effective throughout the event.

Moreover, the system must be highly responsive to real-time conditions. This necessitates that each smart device should compute locally, allowing for point-wise

decision-making. These local analyses can then trigger *collective* actions, such as coordinated movements or emergency evacuations, ensuring that immediate and effective measures are taken to maintain crowd safety.

2.2.3 Autonomous vehicles

In a not-so-distant future where human-driven cars have become a thing of the past, cities will be bustling with *swarms* of autonomous vehicles. These self-driving cars will revolutionize urban transportation, offering a safer and more efficient means of getting from one place to another. Indeed, considering the current situation, where the number of accidents and traffic jams is increasing (only in 2023, there were over 5 million accidents worldwide[1]), this is a very appealing scenario.

However, the complexity of managing these autonomous fleets is far from trivial, especially during peak hours or special events. Take for example Tokyo, where on average it is estimated that over 1 million cars enter the metropolitan area every day[2].

To tackle this intricate challenge, there is a growing emphasis on the need for advanced management systems capable of steering these autonomous vehicles effectively. Each vehicle will be equipped with powerful onboard computers and an array of sensors, enabling them to communicate not only with a centralized traffic management system but also with each other.

The system will operate through continuous data exchange between individual cars and strategically located infrastructure sensors. These sensors monitor various parameters such as traffic flow, road conditions, and even weather. Autonomous cars provide a "street-level" perspective, allowing for real-time adjustments to routing algorithms and speed controls.

This collaborative approach enables the system to pre-empt potential bottlenecks and accidents. For instance, if a major sporting event ends, causing a sudden influx of ride requests, the system can dynamically reroute cars to manage the increased demand efficiently. This proactive approach minimizes congestion and enhances overall traffic flow.

However, the deployment of such a sophisticated system is fraught with challenges. Each autonomous car has a *limited* sensor range and can only offer a *partial* view of the overall traffic landscape. Therefore, advanced machine learning algorithms are essential to integrate this fragmented data into a cohesive and actionable real-time model. Additionally, the system must be capable of *adapting*

[1] `https://www.forbes.com/advisor/legal/car-accident-statistics/`

[2] `https://www.statista.com/statistics/1191368/shutoko-average-daily-traffic-volume/`

quickly to changing conditions, such as road closures, accidents, or even fluctuating demand patterns.

In specific scenarios where immediate action is required–like emergency vehicle passage or sudden road closures–there may be a need to prioritize certain routes or vehicles. Given that each car operates on a finite energy source, the system must *self-organize* to ensure that cars with lower battery levels are routed to charging stations without disrupting the overall traffic flow.

2.3 Characteristics

Drawing from the overview and the diverse examples provided, several distinct characteristics emerge that set CPSWs apart from other large-scale distributed systems. These unique traits not only define the essence of CPSWs but also pose specific challenges in their design and implementation. Each of these characteristics will be discussed in detail below.

Scale CPSWs are inherently scalable, often comprising hundreds or even thousands of interconnected nodes. The behaviour scale independence ensures that the same programmatic logic can be applied across networks of varying sizes, making them highly adaptable to different application scenarios. In doing this, is essential to capture the right collective abstraction that allows both to help the developers to think in terms of the collective behaviour and to effectively design the collective behaviour itself.

Device heterogeneity While the nodes in a CPSW may possess varying hardware capabilities and attributes, they are engineered to operate cohesively. Interoperability stands as a fundamental requirement for these systems, particularly because they often incorporate a diverse array of devices manufactured by different vendors. In today's IoT landscape, semantic interoperability presents a significant challenge. Devices from various vendors might employ divergent communication protocols or data formats, complicating the task of achieving seamless interaction. This heterogeneity is not limited to communication protocols; it also extends to computational power, sensor types, and other attributes. Such diversity introduces an additional layer of complexity in system design, necessitating the identification of a common framework that enables effective communication and collaboration among the devices.

Behaviour homogeneity Despite the diverse hardware configurations, nodes within a CPSW display consistent behaviour at the local level. They run the same local *algorithms*, thereby achieving the system's collective objectives. However,

it is important to clarify that “homogeneous behaviour” does not imply that all nodes do the same thing at the same time. Indeed, given the system’s inherently distributed and extensive: the input received by one node differs from that of another far from it, leading to variations in local behaviour.

Distributed control Centralized control mechanisms are often impractical in CPSWs due to their scale and complexity. In a centralized system, a single controller manages the behaviour of all agents, necessitating high computational power and introducing a single point of failure. This becomes increasingly challenging when dealing with CPSWs that comprise a large scale of highly distributed agents. Any failure or delay in the central controller could lead to system-wide disruptions.

In contrast, distributed control algorithms decentralize the decision-making process, allowing nodes to make local decisions based on their immediate environment and information from neighbouring nodes. This approach offers several advantages:

- **scalability**: as the system grows, the computational burden is distributed across multiple nodes, making it easier to scale;
- **fault tolerance**: since there is no single point of failure, the system can continue to operate even if some nodes fail;
- **adaptability**: distributed control algorithms are often more flexible, enabling the system to adapt to changing conditions without requiring global recalibration;
- **efficiency**: local decision-making can often lead to more efficient utilization of resources, as nodes can make decisions that are optimal for their specific conditions;
- **reduced latency**: decisions can be made more quickly when they are processed locally, which is crucial for real-time applications where delays can have significant consequences.

Self-organization One feature of CPSWs is their ability to *self-organize*. Nodes can dynamically adapt to changing conditions, reconfiguring themselves to maintain system integrity and performance. This is particularly important in scenarios where the environment is unpredictable or hostile. To be more precise, self-organization is defined as a dynamical and adaptive process where systems acquire and maintain structure themselves, without external control. It can be also denoted as a property “to arrange several elements into a purposeful sequential or spatial (or both) order or structure”. Therefore, self-organization is an autonomous, robust, and flexible process that seeks an increase of order [DH04].

Openness CPSWs are typically open systems, meaning that the total node count is not predetermined. New nodes can join or leave the network dynamically, requiring the system to be flexible enough to accommodate these changes without compromising its functionality.

Collective intelligence The nodes in a CPSW work collaboratively to achieve common goals, displaying a form of *collective intelligence* [NKC+09]. This is facilitated by advanced algorithms that enable the system to learn from its environment and adapt its behaviour accordingly. Through decentralized decision-making processes, each node contributes its unique computational resources and sensory data, leading to a more robust and resilient system. This collective approach allows for enhanced efficiency, as the system can dynamically allocate tasks and resources based on real-time needs and constraints. The *collective intelligence* exhibited by the nodes also enables fault tolerance and self-healing capabilities. If one node encounters a failure or is compromised, the remaining nodes can recalibrate and reorganize to continue functioning effectively. Moreover, the collective intelligence is not static; it evolves over time as the system interacts with its environment, making it possible to handle unforeseen challenges and complex scenarios.

Therefore, collective intelligence in CPSW is not merely a by-product but a crucial feature that substantially enhances the system's adaptability, resilience, and overall performance.

Cyber-physical interactions The term "cyber-physical" aptly describes the integration of computational and physical elements in CPSWs. Each node is usually connected to a physical entity via sensors and can influence its environment through actuators. This seamless integration is crucial for applications that require real-time interaction with the physical world, such as industrial automation, healthcare monitoring, and autonomous vehicles.

The sensors collect various types of data– ranging from temperature and pressure to more complex measurements like vibration or chemical composition– which is then processed and analysed by the computational components. These insights enable the nodes to make informed decisions and execute actions via actuators, which might include motors, valves, or other control mechanisms. The end-to-end loop from sensing to actuation ensures a high degree of responsiveness and adaptability, allowing the system to cope with dynamic and unpredictable conditions effectively.

Additionally, cyber-physical integration offers the ability for remote monitoring and control, thus expanding the operational scope and adaptability of the system. For instance, operators can receive real-time data and adjust system parameters without needing to be physically present, thereby reducing both risks and opera-

tional costs.

Furthermore, the cyber aspect of CPSWs often incorporates cloud computing and edge computing solutions. Cloud computing allows for the storage and analysis of large volumes of data, while edge computing provides low-latency processing capabilities closer to the source of data generation. The harmonious blend of cloud and edge computing caters to both large-scale data analytics and real-time processing requirements, thus elevating the system's performance and utility.

In summary, the term "cyber-physical" encapsulates a sophisticated blend of computational intelligence and physical interaction. The seamless integration of these elements requires a modern computing paradigm, making CPSWs highly versatile and robust.

2.4 Related concepts

	MAS	Swarm Robotics	CAS	CPSW
Scale	Dozens	Hundreds	Thousands	Thousands
Capabilities	Heterogeneous	Homogeneous	Heterogeneous	Heterogeneous
Behaviours	Heterogeneous	Homogeneous	Heterogeneous	Homogeneous
Control	Centralized/Distributed	Centralized/Distributed	Distributed	Distributed
Cyber-Physical	Yes/No	Yes	Yes/No	Yes

Table 2.1: Summarized comparison between MAS, Swarm Robotics, CAS and CPSW

This section will discuss some of the related concepts that are closely aligned with CPSWs summarized in Figure 2.2 and Table 2.1.

Multi-Agent Systems In the realm of computational systems, a CPSW can be closely related to a Multi-Agent System (MAS), with a specific emphasis on being a *many*-agent system. In a MAS, multiple autonomous agents interact with each other to achieve specific objectives. These agents are capable of sensing their environment, making decisions based on their observations, and then acting upon those decisions. The key difference in a CPSW is the scale and the integration of physical components, such as sensors and actuators, which allows for real-time interaction with the environment.

In a CPSW, agents are not just virtual entities but are tethered to physical components, making them cyber-physical agents. These agents continuously engage in a cycle of *repeated* sensing, computation, communication, and actuation to achieve collective behaviours.

One of the significant challenges in designing a CPSW is the high stochastic of the environment. Unlike controlled settings where outcomes can be predicted

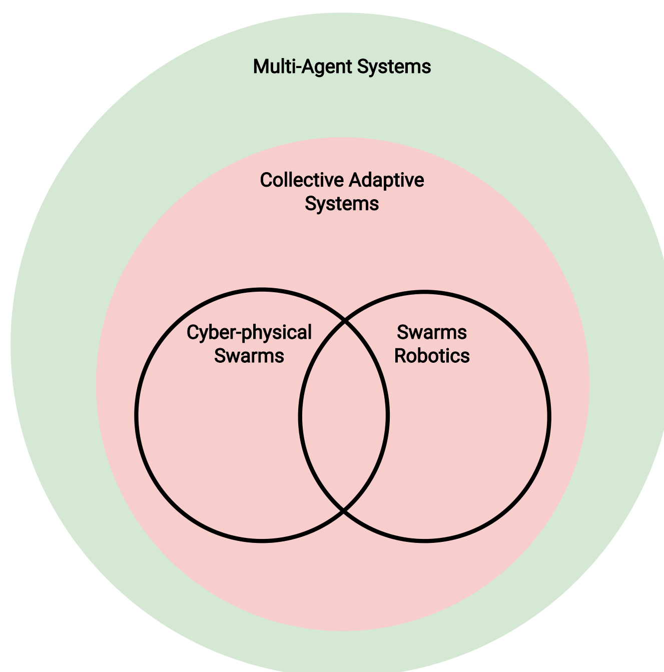


Figure 2.2: Taxonomy of Cyber-Physical Swarm (CPSW) w.r.t related Systems

with a high degree of accuracy, CPSWs often operate in dynamic and unpredictable environments. This makes it nearly impossible to pre-program optimal behaviours for all agents. The agents must, therefore, be capable of adapting their behaviours in real time based on their sensory inputs and the inputs from other agents in the network. This requires sophisticated algorithms that can handle uncertainty and make near-optimal decisions in real time.

Moreover, the agents in a CPSW are designed to work collaboratively to achieve collective goals, rather than pursuing individual objectives. This is in contrast to some MAS where agents might have conflicting goals. In a CPSW, the focus is on achieving a form of collective intelligence through distributed computing and decision-making. Agents share information and coordinate their actions to solve complex problems that are beyond the capabilities of any single agent.

In summary, while CPSWs share similarities with MAS in terms of multi-agent interaction and decision-making, they extend the concept by incorporating large-scale, cyber-physical integration, and a focus on collective intelligence. The challenges in designing CPSWs are manifold, ranging from handling the stochastic nature of the environment to ensuring robust and adaptive collective behaviours.

Swarm intelligence It is a branch of collective intelligence with an interdisciplinary field that draws inspiration from the collective behaviours observed in

social animals, such as ants, birds, and fish, to develop computational algorithms and systems. Initially, the focus was primarily on swarm robotics, where the objective was to create robotic systems that could emulate the complex behaviours seen in natural swarms. The methodology employed is fundamentally bottom-up. Designers and researchers study the behaviours of individual animals in their natural habitats to understand the rules or heuristics they follow. These individual behaviours are then modelled computationally to observe how they contribute to the emergence of collective intelligence in a swarm. One seminal concept that emerged from this line of inquiry is *stigmergy* [DBT00]. Stigmergy is a form of indirect communication and coordination where agents in a swarm interact with each other by modifying a shared environment, rather than through direct communication.

In recent years, the focus of swarm intelligence has evolved to concentrate more on algorithmic aspects. Researchers have started to leverage the principles of collective behaviour to develop optimization algorithms that can solve complex computational problems. These algorithms are particularly useful in scenarios where traditional optimization methods are computationally expensive or fail to find optimal solutions within a reasonable time frame. Some of the most well-known algorithms that have emerged from Swarm Intelligence research include Ant Colony Optimization (ACO) [DMC96], Particle Swarm Optimization (PSO) [KE95], and Flock of Starling Optimization (FSO) [FS11]. Each of these algorithms has its own set of rules and heuristics, modelled after the specific animal behaviours they are inspired by, and they have been applied successfully in various domains such as network design, resource allocation, and data clustering.

While these optimization algorithms offer valuable methodologies and have broad applicability, they are not the central focus of this thesis. Differently, the focus is on harnessing the *principles* of swarm intelligence to develop *artificial* systems that can achieve similar collective behaviours through mechanisms of *self-organization*. Unlike traditional swarm intelligence algorithms that often aim to *mimic* nature, this thesis draws *inspiration* from natural systems to understand the fundamental principles that make these systems *robust*, *scalable*, and *adaptable*. The ultimate goal is to exploit these principles to design and implement artificial swarm systems that can operate effectively and efficiently in complex, dynamic, and potentially hostile environments.

Swarm robotics Historically, the field of swarm robotics has its roots in the early approaches to swarm intelligence. Over time, it has evolved to become the *engineering* arm of swarm intelligence, focusing on the *practical* aspects of building and maintaining swarm systems. The overarching goal of *swarm engineering* is to establish a rigorous methodology for the entire lifecycle of a swarm robotics system, from conceptualization to operation and maintenance [Bra+13].

In traditional swarm robotics, the primary focus is on *robots* that are *autonomous*, *situated*, and operate under *no central control*. These robots are designed to interact with each other and their environment to achieve collective goals. CPSW extends this paradigm to include other "swarm-like" systems that may not necessarily involve robots. Examples include crowds of people in public spaces, large-scale IoT networks, and smart city infrastructures. These systems share many similarities with swarm robotics, such as the need for autonomy, localized decision-making, and collective behaviour, but they also present unique challenges and opportunities.

One of the most promising avenues for extending the principles of swarm robotics to these other domains is the emerging field of *automatic design* [FB16]. In automatic design approaches, the control logic for the agents in the swarm is not manually programmed but is instead derived through optimization techniques such as *genetic algorithms* or *multi-agent reinforcement learning*. These methods aim to optimize a *global* utility function that captures the overall objectives of the swarm. This is particularly appealing for CPSWs, where the complexity and scale of the system make manual programming impractical.

In summary, the principles and methodologies developed in swarm robotics provide a strong foundation for the engineering of CPSWs. However, the unique challenges and complexities of CPSWs require further innovation. By leveraging the principles of swarm intelligence, we can create a more robust, adaptable, and intelligent CPSWs that can operate effectively in a wide range of applications and environments.

Collective adaptive systems Collective adaptive systems (CAS) are a broad class of systems composed of agents (potentially heterogeneous) capable of adapting to environmental conditions while striving to achieve a collective goal through the emergence of individual node cooperation. Typically, individual units are simple and draw strong inspiration from natural systems. In these systems, which are adaptive by nature, there is interest in incorporating *self-** properties (in fact, these CAS are sometimes discussed as collective self-adaptive systems). These properties include *self-healing*, *self-optimization*, and *self-configuration*. Self-healing refers to the system's ability to recover from failures without human intervention. Self-optimization means the system can improve its performance over time based on feedback. Self-configuration allows the system to adapt to changing conditions without requiring manual adjustments. These self-* properties contribute to the system's overall *resilience* and *efficiency*.

CPSW can be understood as a specialized subset of the broader CAS. This specialization arises from two key distinguishing characteristics: i) the involved entities are not merely digital but also have a *cyber-physical* nature, integrating

both computational and physical elements, and ii) despite the heterogeneous nature of the devices within the system, their behaviour manifests uniformly. These unique attributes have a profound impact on the system's design methodology, which is why this thesis specifically focuses on this subclass of systems.

2.5 Final Remarks

Cyber-Physical Swarm (CPSW) represent a specialized subset of Collective Adaptive Systems (CAS), distinguished by their cyber-physical nature and homogeneous behaviour despite device heterogeneity. Drawing from the principles of swarm intelligence, multi-agent systems, and swarm robotics, CPSW offer a promising avenue for tackling complex, large-scale problems in a variety of domains, from environmental monitoring to crowd management and autonomous transportation.

The unique characteristics of CPSW, such as scale, device heterogeneity, behaviour homogeneity, and self-organization, not only define their essence but also pose specific challenges and opportunities in their design and implementation. These challenges necessitate innovative approaches in automatic design, distributed control, and self-organization.

This thesis aims to contribute to the understanding and engineering of CPSW by exploring these challenges and proposing methodologies that leverage the principles of collective behaviour to design systems that are robust, scalable, and adaptable.

Chapter 3

Macro-programming

What is macroprogramming?
Why is it important?
What is aggregate computing?
What is the *execution model* of aggregate computing?
– **RQ1, RQ4**

Contents

3.1 The Essence of Macroprogramming 25
3.2 Conceptual Framework 26
3.2.1 Preliminaries 26
3.2.2 Macroprogramming: Definition and Basic Concepts . . 26
3.2.3 Historical Evolution and Context 27
3.3 Aggregate computing 28
3.3.1 System Model 29
3.3.2 Field-based Programming Model 31
3.3.3 Tools 40
3.4 Final Remarks 47

Macroprogramming, as a paradigm, has emerged as a pivotal approach in the realm of large-scale distributed systems. It offers a unique perspective, allowing developers to express the *macroscopic behaviour* of a collective system. This chapter delves into the intricacies of macroprogramming, its historical evolution, and its significance in the modern computational landscape, leading to aggregate computing–a novel macro-programming approach for programming collective self-organizing behaviours in highly scalable and distributed systems.

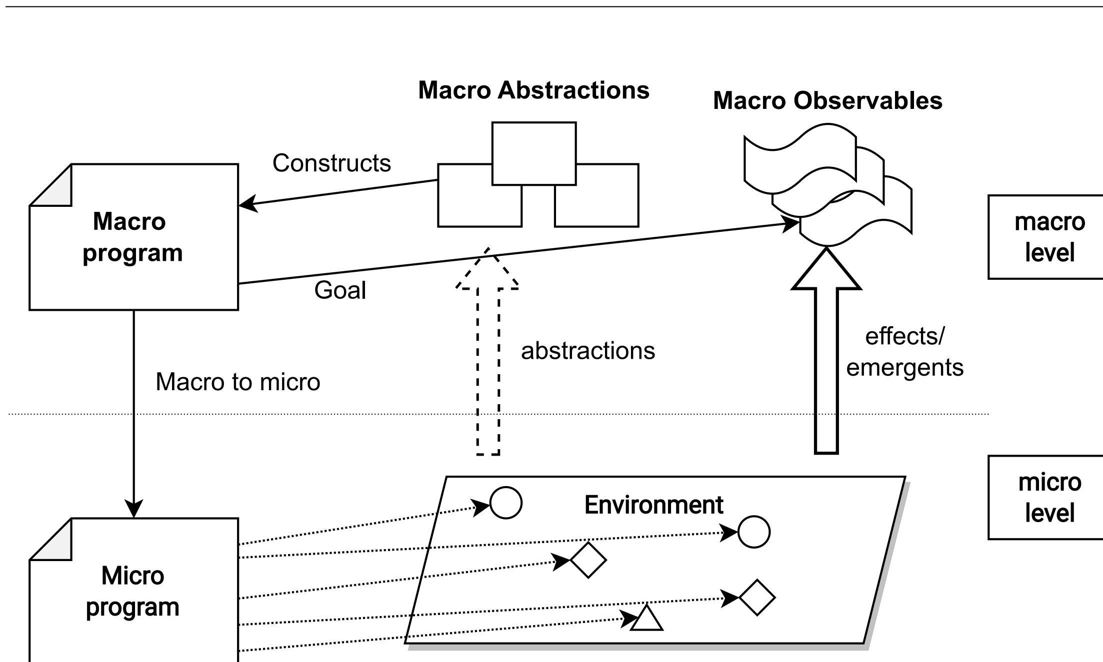


Figure 3.1: Overview of macroprogramming, inspired from [Cas23]

3.1 The Essence of Macroprogramming

Macroprogramming [Cas23] emphasizes the overarching behaviour of a system, abstracting the intricacies of individual components. The primary motivation behind macroprogramming is to simplify the design and development of collective systems. By providing a higher-level perspective, it allows developers to address system-wide concerns without getting buried by the details of individual components. Indeed, developers should focus solely on the system's overall behaviour, rather than worrying about its technological details, individual component capabilities, or communication protocols. This not only streamlines the development process but also ensures that the system's behaviour is *consistent* and *predictable*.

In essence, macroprogramming is about seeing the *forest* for the *trees*. It recognizes that while individual components (or trees) are essential, understanding and managing their collective behaviour (or the forest) is of paramount importance. This perspective is particularly relevant in today's interconnected world, where systems often comprise numerous components that need to work in harmony.

Furthermore, macroprogramming leverages macro-level abstractions, such as *collective states*, *groups*, or *spatiotemporal* patterns. These abstractions provide a structured way to think about and design systems, ensuring that they are both *robust* and *adaptable*. By focusing on these higher-level abstractions, macroprogramming allows for a more intuitive and efficient approach to system design,

making it an indispensable tool in the modern developer's toolkit. Figure 3.1 provides an overview of the idea behind this paradigm.

3.2 Conceptual Framework

3.2.1 Preliminaries

Macroprogramming addresses the challenge of programming the behaviour of a computational system S, composed of multiple computational entities. Given two entities A and B within this system, there are three primary modes to influence their behaviour to promote properties ascribable to S:

1. altering their context, indirectly influencing them—e.g., a change in sensor A might subsequently affect B;
2. interaction, such as triggering their behaviour—e.g., if A is an actuator, its actions might influence B;
3. setting their behaviour to produce certain global outcomes when activated.

The term "program" refers to an abstract description executable by a computational entity. Modes (1) and (2) allow an external entity C to influence A or B, and consequently S.

3.2.2 Macroprogramming: Definition and Basic Concepts

Macroprogramming is defined as an abstract paradigm for programming the macroscopic behaviour of systems of computational entities. As a paradigm, it is an approach rooted in a mathematical theory or a set of coherent principles. The foundational principles of macroprogramming include:

- *micro-macro distinction*: recognizing two primary system levels – macro (global structures) and micro (computational entities);
- *macroscopic perspective*: focusing on the system's macroscopic aspects, considering micro-level entities from a global perspective;
- *microprogram*: the result of macroprogramming deployment is a program executed by individuals reaching the macroscopic perspective;
- *macro-to-micro mapping*: implementing how a macro program is executed by the system, defining a logic to map macro instructions to micro-level behaviours.

Particularly, the concept of micro-macro levels is common in various scientific areas including social sciences and artificial intelligence. In programming, particularly in macroprogramming, the micro and macro dimensions are defined by specific design boundaries. Micro entities have computational behaviour and can be autonomous, active, or reactive. They interact with other micro-entities and can include agents, actors, objects, and microservices. Macro entities can be categorized as macro-level observables and macro-level constructs. Macro-level observables are high-level system behaviours that may be hard to deduce from micro-level states. The goal of a microprogram is generally a function of these macro-level observables. Macro-level constructs are abstractions that can affect the behaviour of two or more micro-level entities. The challenge in macroprogramming is to effectively map these macro-level constructs to micro-level entities. The notion of 'emergence' is key to understanding the relationship between the micro and macro levels.

Another essential aspect is the concept of "collectives", which are groups of similar entities that share common traits, goals, or mechanisms for interaction. Examples of collectives include a group of co-located workers or a swarm of drones. These entities within a collective usually share common goals and interaction mechanisms, although differences among them can enhance collective capabilities. Heterogeneous collectives, which include diverse elements like humans, robots, and sensors, also exist and can be managed through macroprogramming. However, heterogeneity complicates macroprogramming by emphasizing individual perspectives and widening the gap between macro-level and micro-level considerations.

Finally, macroprogramming often employs a declarative approach, focusing on specifying what the goal of a computation is rather than detailing how to achieve it. Declarative programming allows for the abstraction of elements like function evaluation order, theorem proving, and data access specifics. The aim is to offer high-level abstractions that capture system-wide concerns and can be conveniently mapped to component-level issues. Because these assumptions are often specific to a particular application area, macroprogramming languages usually manifest as Domain-Specific Languages (DSLs).

3.2.3 Historical Evolution and Context

The historical roots of macroprogramming can be traced back to the pioneering work of Newton and Welsh [NW04]. Their research laid the foundation for the application of macroprogramming in the domain of Wireless Sensor Networks (WSNs). These networks, characterized by embedded units equipped with processing, communication, and sensing capabilities, presented unique challenges that necessitated a macroscopic view for effective data processing and logic description. The emphasis was on capturing the collective behaviour of these sensor nodes, ensuring efficient data aggregation, processing, and communication. WSNs were

among the first systems that required a departure from traditional programming paradigms. Given the distributed nature of these networks and the limited resources of individual sensor nodes, there was a pressing need to optimize both computation and communication. Macroprogramming emerged as a solution, allowing developers to focus on the overall behaviour of the network rather than the intricacies of individual nodes.

As technology evolved, so did the applications of macroprogramming. The rise of the Internet of Things (IoT) in the subsequent years brought forth a plethora of interconnected devices, each with its own set of capabilities and functions. The complexity of these systems further underscored the importance of a macroscopic approach. Macroprogramming principles were adapted and refined to cater to the diverse requirements of IoT ecosystems. Furthermore, the emergence of Cyber-Physical Systems (CPSs) and *spatial computing* [Bea+12] introduced new challenges and opportunities for macroprogramming. These systems, which integrate computational processes with physical entities, demanded a holistic approach to ensure seamless interaction and coordination. Macroprogramming, with its emphasis on collective behaviour and high-level abstractions, proved to be an invaluable tool in this context. This research trend culminates with the advent of *aggregate computing* [BPV15], a novel macroprogramming approach for programming collective self-organizing behaviours in highly scalable and distributed systems.

3.3 Aggregate computing

Field-based coordination [Vir+19] is an approach where computation leverages a notion of *computational fields* (*fields* for short) [War89; MZL04; Vir+19], namely, distributed data structures evolving in time and associating locations with values. The approach originates from previous work like Warren's *artificial potential fields* [War89] and *co-fields* from Mamei et al. [MZL04]. In particular, in co-fields, computational fields represent contextual information, locally sensed by the agents and repeatedly distributed by the agents themselves or the infrastructure according to a propagation rule.

In this work, by field-based coordination we mean a specific programming and computational model, also known as *aggregate computing* in literature [BPV15], which is surveyed in [Vir+19]. In this model, collective and self-organising behaviour is programmed through a composition of functions operating on fields mapping a set of individual agents (rather than environment locations) to computational values. Therefore, fields can be used to associate a certain domain of agents with what they sense, the information they process, and actuation instructions for operating in the environment. Fields are computed locally to the agents but are subject to a global viewpoint: so, e.g., a field of velocity vectors can be seen

as a movement command for an entire swarm, or a field of double can denote what an entire swarm perceives in a certain environment. To understand field-based computing, two essential parts have to be considered: the *system model* and the *programming model*. Their interplay is what allows the local actions of the agents to yield emergent collective behaviour.

3.3.1 System Model

We consider a network of computing and interacting *agents* situated in some *environment*, compatible with the vision of Cyber-Physical Swarm (CPSW), therefore considering the behavioural homogeneity and the cyber-physical aspect.

Structure. An *agent* is an autonomous entity equipped with *sensors* and *actuators*, which serve as the interface towards a logical or physical *environment*. From a logical point of view[1], it also has *state*, a support for *communicating* with other agents, and support for *computing* simple programs. An agent is connected with other *neighbour* agents which collectively form its *neighbourhood*. The set of neighbours depends on a *neighbouring relationship*, which is defined by designers according to the application at hand and is subject to the constraints exerted by the underlying physical network. A typical neighbouring rule is the one that mimics physical connectivity; so, e.g., a robot is a neighbour of another robot if it manages to send a message to the latter over the wireless channel. Another typical neighbouring rule is the one based on spatial vicinity; so, e.g., a robot is a neighbour of another robot if the infrastructure manages to deliver a message from the former to the latter (e.g., using other robots as relays) and these two robots are at an estimated distance smaller than a certain threshold (assuming a distance can be estimated through proper technology).

Interaction Interaction happens by sending messages to neighbours, *asynchronously*. Interaction can also happen in a stigmergic way, by perceiving and acting upon the environment through sensors and actuators. The content of messages and when they are sent and received depends on the agent's behaviour. However, in general, as our goal is to model continuous collective behaviours or self-organising systems, we remark that interaction would typically be frequent (in relation to the problem and environment dynamics).

[1] Actually, such requirements may be relaxed by considering different execution strategies on available infrastructure [Cas+20b].

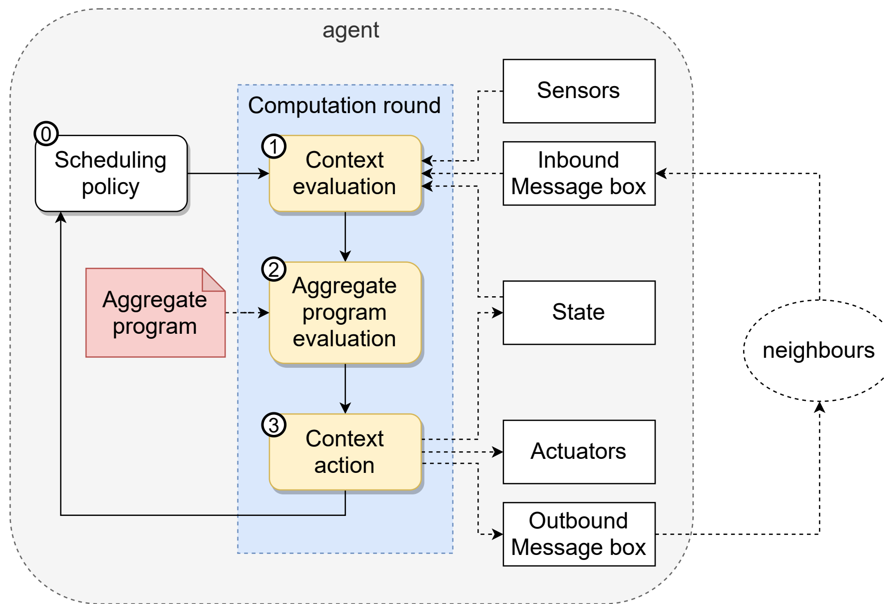


Figure 3.2: High-level behaviour of an agent in an aggregate system, taken from [CAV21b]

Behaviour As per the above consideration, the behaviour of any individual agent is best understood in terms of repeated action of *execution rounds* (see Figure 3.2), where each round consists of the following steps (though some flexibility exists especially in the actuation part):

1. *Context acquisition.* The agent gathers its context by considering its previous state as well as the most recent sensor readings and messages from neighbours.

2. *Computation.* The agent runs a computation against the acquired context, yielding (i) an *output* describing potential actions; and (ii) a *coordination message* containing all the information to be sent to neighbours for coordination at a collective level.

3. *Actuation and communication.* The agent performs the actions described by the program output and dispatches the coordination message to the entire neighbourhood.

By having every agent repeatedly run these sense-compute-act rounds, the whole system fosters a self-organization process whereby up-to-date information (from the environment and from the agents) is continuously incorporated and processed, typically in a self-stabilising manner [Dol00].

$\mathtt{P} ::= \mathtt{F}\ \mathtt{e}$	program
$\mathtt{F} ::= \mathtt{def}\, \mathtt{f}\,(\bar{\mathtt{x}})\{\mathtt{e}\}$	function declaration
$\mathtt{e} ::= \mathtt{x} \mid \mathtt{v} \mid \mathtt{f}(\bar{\mathtt{e}}) \mid \mathtt{if(e)\{e\}\{e\}} \mid \mathtt{nbr\{e\}} \mid \mathtt{rep(e)\{(x)} \rightarrow \mathtt{e\}}$	expression
$\mathtt{f} ::= \mathtt{d} \mid \mathtt{b}$	function name
$\mathtt{v} ::= \mathtt{l} \mid \phi$	value
$\mathtt{l} ::= \mathtt{c}(\bar{\mathtt{l}})$	local value
$\phi ::= \bar{\delta} \mapsto \bar{\mathtt{l}}$	neighbouring field value

Figure 3.3: Field Calculus abstract syntax extracted from [Vir+18a].

This system model provides a basic machinery for collective adaptive behaviour, which however requires a proper description of the "local computation step": this is fostered by the *field-based programming model* (discussed in Section 3.3.2). A *field-based program* steers the collective adaptive behaviour of a system, which unfolds by having each agent in the system evaluate that program according to the discussed round-based execution model. Notice that such a program specifies both what local processing the agents must perform and what data they must share with neighbours; also, notice that generally, the program does not affect the round-based execution protocol—unless advanced forms of scheduling are desired [Pia+21; ACV22a]. The distributed execution protocol may be provided by a *middleware*, which will ensure that messages are exchanged and rounds properly scheduled. The reader can refer to [Pia+21] and [Cas+22], respectively, for a more comprehensive discussion on execution and deployment aspects.

3.3.2 Field-based Programming Model

Field Calculus (FC) was originally conceptualized in [VDB13] as a foundational framework aimed at capturing essential elements in computational field languages. These elements include field function definitions, functional interplay with fields, time-based field changes, the building of field values from surrounding nodes, and limiting computations to network sub-regions.

One of the distinctive aspects of field calculus is its dual interpretive nature. On a local scale, the specification describes cyclic computations on individual devices (i.e., the system model described above). Conversely, at the global level, a field calculus expression maps each computational round for every device to its corresponding space-time value. This inherent duality effectively bridges the gap between individual device behaviour and the emergent global network behaviour, a

claim supported by computational adequacy and abstraction properties as outlined in [Aud+19a].

Figure 3.3 describes the abstract syntax for field calculus. In this syntax, an overbar notation ($\bar{\mathtt{e}}$) indicates a sequence of elements, namely $\bar{\mathtt{e}}$ stands for $\mathtt{e}_1, \mathtt{e}_2, \ldots, \mathtt{e}_n$ for some $n > 1$. When multiple overbar notations are used in the same expression, they are assumed to have the same length and they are expanded together, namely $\bar{\delta} \mapsto \bar{\mathtt{l}}$ stands for $\delta_1 \mapsto \mathtt{l}_1, \delta_2 \mapsto \mathtt{l}_2, \ldots, \delta_n \mapsto \mathtt{l}_n$ for some $n > 1$. Keywords in this syntax include `def` for function definitions, `if` for conditional expressions, and `rep` and `nbr` for specific field calculus operations related to time-based state evolution and neighbour data sharing, respectively.

A typical field calculus program `P` comprises a series of function declarations $\bar{\mathtt{F}}$ followed by a main expression `e`, which collectively describes both global and local system behaviour. An expression `e` can be:

- a *variable* `x`;
- a *value* `v`, which can be either:
  - a *local value* `l`, defined through a constructor `c` applied to a sequence of arguments $\bar{\mathtt{l}}$, such as boolean, number, string, etc.;
  - a *neighbouring field value* ϕ, defined through a sequence of pairs $\bar{\delta} \mapsto \bar{\mathtt{l}}$, where $\bar{\delta}$ is a sequence of neighbouring device (including itself) and $\bar{\mathtt{l}}$ is a sequence of local values, e.g., a map of neighbouring devices to the distance from them;
- a function application $\mathtt{f}(\bar{\mathtt{e}})$, where `f` is a function name and $\bar{\mathtt{e}}$ is a sequence of expressions, e.g., $\mathtt{f}(\mathtt{x}, \mathtt{v})$. It can be *user defined*, e.g., `d(x)`, or *built-in*, e.g., `b(v)`;
- a *branching expression* `if(e){e}{e}`, where `e` is a boolean expression and `e` is an expression, e.g., `if(b(v)){d(x)}{v}`;
- a *neighbourhood expression* `nbr{e}`, which creates a neighbouring value mapping neighbours to their latest available result of evaluating e. In particular, each device δ:
  - shares its value of `e` with its neighbours, and
  - 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;
- A *neighbourhood expression*, denoted as `nbr{e}`, generates a mapping that associates neighbours with their most recent evaluation results of `e`. Specifically, for each device δ:

- It disseminates its evaluation of `e` to its neighbours;
- It computes the expression into a neighbourhood value function Φ. This function Φ maps each neighbouring device δ' to the most recent evaluation of `e` received from δ;

For example, `nbr(humidity())` (where `humidity()` is a built-in sensor estimating local humidity) would result in a neighbourhood value function Φ, which maps each neighbour to the humidity level measured by that neighbour. It is worth noting that in an `if` statement, sharing is confined to devices within the same branch's subspace. This is because devices in different subspaces do not execute the same `nbr(e)` constructs;

- `rep(e){(x)` $\to$ `e}`, where `e` is an expression, `x` is a variable, and `e` is an expression. It simulates the dynamic evolution of the state over time intervals. This construct extracts the previously computed value v from the complete `rep` expression during the last evaluation cycle. For the initial evaluation, the expression `e1` is evaluated to provide the starting value of v. In subsequent rounds, v is updated based on the outcome of evaluating `e2`, where each instance of x is substituted with v.

Within this suite of operations, the `nbr` and `rep` constructs serve distinct roles, facilitating message exchanges between devices and managing states within the iterative rounds of an individual device, respectively. These constructs are underpinned by a data gathering mechanism enabled through a technique known as *alignment* [Aud+16]. This ensures accurate message correspondence, eliminating the possibility of unintended message swapping between different instances of `nbr` expressions, as well as preventing state memory interchange between different instances of `rep` expressions. A significant implication of this is the isolated execution of the two branches in an `if` statement within field calculus; a device executing the `then` branch is unable to communicate with the `else` branch of a neighbouring device, and vice versa.

3.3.2.1 Examples

The proposed model and language are quite simple and intuitive, yet it is expressive enough to capture a wide range of collective behaviours. In the following, we present a few examples of collective behaviours that can be expressed in this model.

Direct neighbour interaction (space) In the realm of aggregate computing, one can effectively calculate spatial structures by utilizing the `nbr` construct. Consider, for example, the task of computing the average temperature perceived by a

device along with its neighbouring devices. This can be seamlessly implemented through the `nbr` construct as shown below:

```
val totalTemperature = foldhood(0.0)(_ + )(nbr{temperature()})
val neighboursCount = foldhood(0)( + _)(nbr{1})
val averageTemperature = totalTemperature / neighboursCount
```

Here, `foldhood` is a built-in function designed for the aggregation of neighbour values. It takes three parameters:

1. the neutral element, which serves as the initial value for the aggregation;
2. the aggregation function, which defines how the values should be combined;
3. the query to neighbours, which specifies what information is to be collected from each neighbour.

Local field evolution (time) If with `nbr` we can collect information from the neighbourhood, with `rep` we can evolve the state of the device over time. For instance, consider the following example:

```
rep(0) { local => local + 1}
```

This expression initializes the state of the device to 0 and then increments it by 1 in each subsequent round, namely, it simulates a counter. Combining these two operators, it is possible to create space-time patterns, such as the following, that simulates a wave propagating in space. One such pattern is the self-healing gradient.

Self-healing gradient (space-time) A gradient is a field that associates each device in the system with its shortest distance to the nearest source device. A *self-healing* gradient algorithm calculates this gradient field and autonomously updates it to reflect alterations in the source set or network connectivity. This algorithm is significant because it frequently serves as a component in more complex self-organizing algorithms, such as those used for managing information flows, gathering distributed data, and segmenting networks into regions. Using the field calculus, this can be expressed as follows:

```
// distance from source region with nbrRange metric
def distanceTo(source) {
    rep (Infinity) {
        (dist) =>
            mux (source) { 0 }
            { minHood(nbr{dist} + nbrRange()) }
    }
}
```

where mux is a conditional expression, minHood is a built-in function that returns the minimum value in the neighbourhood, and nbrRange is a built-in function that returns neighbouring field distance.

3.3.2.2 Aggregate programming stack

Building upon both theoretical insights and pragmatic considerations, aggregate programming introduces a stratified architecture designed to significantly ease the design, development, and maintenance of intricate distributed systems. This methodology stems from three pivotal observations related to engineering sophisticated coordination schemas:

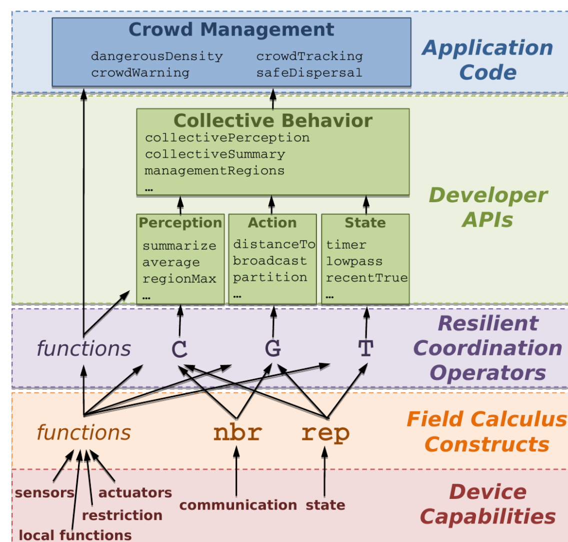
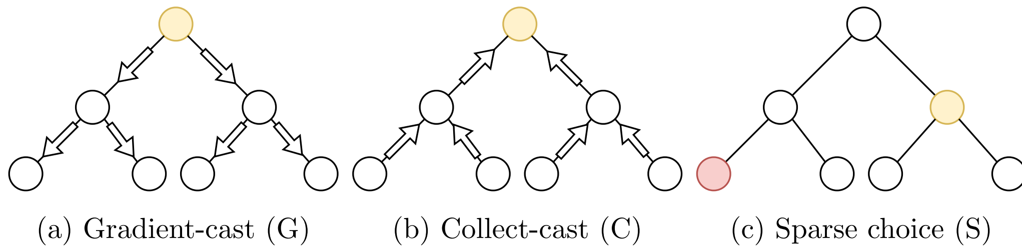


Figure 3.4: The aggregate programming stack taken from [BPV15]

- the composition of modules and subsystems should be straightforward and transparent;
- various subsystems necessitate distinct coordination mechanisms that are context-sensitive, varying according to regions and temporal conditions;
- robust coordination mechanisms should be encapsulated within abstractions, thereby obviating the need for programmers to grapple with their underlying complexities.

Field calculus, along with its language incarnations, offers solutions for the first two observations but falls short of ensuring resilience. Additionally, its mathematical rigour and concise syntax present challenges for straightforward programming. Consequently, supplementary methodologies are indispensable for scaling effectively with system complexity. These pattern have been identified in literate [BPV15] and are called resilient coordination operators (more details in Section 3.3.2.3 and Section 3.3.2.4). Finally, on top of these operators, there are high-level patterns that can be used to implement complex behaviours (more details in Section 3.3.2.5) and high-level API for structuring domain-specific behaviour (e.g., crowd detection or coordinate movement in swarms). The overall stack is shown in Figure 3.4.



(a) Gradient-cast (G) (b) Collect-cast (C) (c) Sparse choice (S)

Figure 3.5: Resilient coordination operators

3.3.2.3 Resilient coordination operators

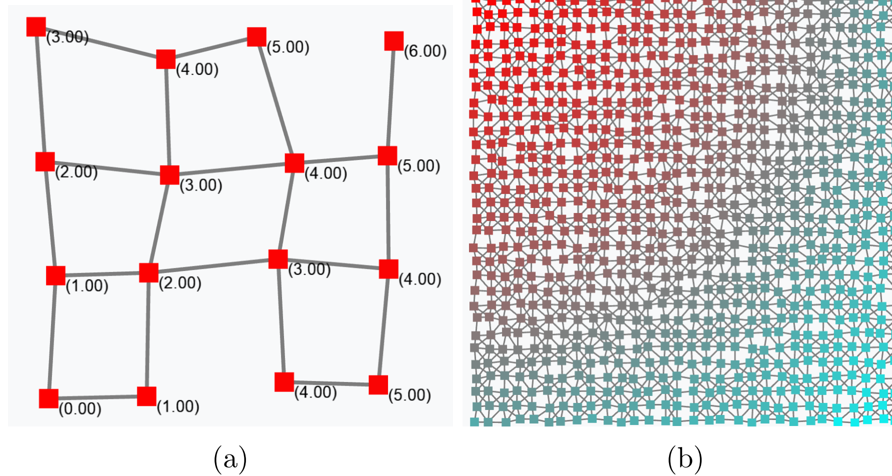
Throughout the development of aggregate computing, a consistent set of foundational building blocks has emerged over the years (Figure 3.5). These building blocks are known for their self-stabilizing properties–details on which will be discussed later–and they serve as the foundation for various high-level patterns in aggregate computing.

Gradient-cast (G) — Figure 3.5a various forms of aggregation can be executed along a distance gradient. This algorithm, called G, is tailored based on a specific *metric* used for measuring increments or distances. It enables the transfer of a particular field value from the source in an outward direction, evolving according to a specified logic as it ascends the gradient:

```
def G(src, phi, acc, metric) =
    rep((Double.MaxValue, phi)) { case (distance,value) => {
        mux(src) {
            (0.0, phi)
        } {
            minHoodPlus { (nbr(distance) + metric, acc(nbr(value))) }
        }
    }
}._2
```

Where `minHoodPlus` is a built-in function that returns the minimum value in the neighbourhood, and `nbr` is a built-in function that returns neighbouring field distance. The `_._2` at the end of the expression is used to extract the second element of the tuple. Figure 3.6 shows the gradient computation in a sparse and dense network of devices. This operator serves as the foundation for a range of gradient-based algorithms, such as “broadcast”. In the broadcast method, data is disseminated along the gradient path:

```
def broadcast(source, field) =
    G(source, field, acc = x => x, metric = nbrRange)
```



(a) (b)

Figure 3.6: Gradient computation in a sparse and dense network of devices. In Figure 3.6a, the output from executing a gradient program is presented, with the source node positioned in the bottom-left corner. Figure 3.6b displays the execution of a gradient program in a dense network, where the source node is situated in the top-left corner. The intensity of the colour serves as a gauge for proximity, with redder hues indicating closer distances.

Collect-cast (C) — Figure 3.5b in essence, G facilitates the flow of information from originating devices to their broader environment, acting as a mechanism for the spread or diffusion of values. Conversely, a secondary operation enables data to flow from a widespread area to specific gathering points, aiding in decentralized sensing tasks. This is enabled by another generalized operator C – it accumulates values along a potential field, getting the data from the lowest point of the gradient up to the highest point, that is the centre of the potential field:

```
def C(potential, acc, phi, Null) = {
  rep(phi) { v =>
    acc(phi, foldhood(Null)(acc) {
      mux(nbr(findParent(potential)) == mid()) {
        nbr(v)
      } {
        nbr(Null)
      }
    })
  }
}

def findParent(p) = {
  mux(minHood { nbr(p) } < p) {
    minHood { nbr { (p, mid()) } }._2
  } {
    Int.MaxValue
  }
}
```

Sparse choice (S) — Figure 3.5c the generic operator Sparse choice (S) allows for the selective inclusion of devices to divide the network into distinct "responsibility zones". Essentially, it performs a leader election procedure. In this process, a "grain" represents the average distance between two elected leaders, as defined by a specific metric. It can be implemented as follows:

```
def S(grain, metric) = breakUsingUids(randomUid, grain, metric)

def breakUsingUids(uid, grain, metric) =
    uid == rep(uid) { lead =>
      val acc = (_) + metric
      distanceCompetition(G(uid == lead, 0, acc, metric), lead, uid, grain, metric)
    }

def distanceCompetition(d, lead, uid, grain, metric) = {
  val inf = (Double.PositiveInfinity, uid._2)
  mux(d > grain) { uid }
  {
    mux(d >= (0.5 * grain)) { // 0.5 is a constant used to avoid oscillations
      inf
    } {
      minHood {
        mux(nbr { d } + metric >= 0.5 * grain) {
          nbr { inf }
        } {
          nbr { lead }
        }
      }
  }}}
```

3.3.2.4 Behavioural Properties

The field calculus is architected as a universal language specifically tailored for computations in spatially distributed systems. One of the most critical properties studied within subsets of this core language is that of *self-stabilization*, which ensures the system's ability to autonomously reach a correct state [LLM15].Defined formally within the context of the transition system $N \xrightarrow{act} N$ representing network evolution, self-stabilization guarantees that:

1. the program evaluation, given an eventually constant input, will converge to a stable value at each device within a finite timeframe;

2. this stable value is solely determined by the current input values, thus mitigating any interference from transitory states.

Self-stabilizing algorithms are particularly robust in dynamically evolving systems, reacting coherently to input changes without residual influence from prior states.

The study in [DV15] identifies an initial set of self-stabilizing fragments through a 'spreading operator', which performs monotonic updates on neighbouring values through a diffusion function. These fragments allow for versatile combinations with

local operations, excluding explicit `rep` and `nbr` expressions. Nevertheless, this framework supports several fundamental building blocks like distance estimation and broadcast.

An expanded set of self-stabilizing fragments and building blocks are covered in [Vir+18b]. This work proposes usage restrictions on `rep` statements to specific patterns: *converging*, *acyclic*, and *minimising*. These correlate with the three primary building blocks: G, C, and T, each with distinct functionalities and applications. Moreover, the study delves into the notion of 'equivalence and substitutability' for self-stabilizing programs. This concept not only allows for program optimization through the substitution of more efficient equivalents but also provides a new lens for understanding self-stabilizing programs by abstracting their transient behaviours.

The work establishes different semantic interpretations of a given program: operational semantics (local viewpoint), denotational semantics (global viewpoint), and eventual behaviour (limit viewpoint). Another perspective, termed as the 'continuous viewpoint,' is explored in [Bea+17]. This involves the convergence of output values towards a continuous function, as the density of computing devices in an area increases.

Taking cues from self-stabilization principles, the notion is relaxed to define 'eventually consistent' programs. These programs are expected to converge to a limit continuously, barring a transient initial period, assuming that the inputs remain constant beyond this initial period. This eventual consistency is demonstrated for all programs expressible in the GPI calculus [Aud+18].

Lastly, contemporary work is beginning to examine the real-time performance guarantees of field calculus programs [Aud+18]. Current validation approaches primarily focus on 'by construction' proofs based on elementary building blocks or restricted fragments of the calculus.

3.3.2.5 High-level patterns

Starting from the previously defined building blocks, it is possible to define high-level patterns that can be used to implement complex behaviours. In the following, we illustrate two of the most idiomatic in the context of aggregate computing, which are the *self-healing channel* and the *self-organising coordination* regions pattern.

Self-healing channel The self-healing channel pattern is a distributed computation that produces a path between two points in the network. This path is resilient to the failure of intermediate nodes, and it is dynamically updated when the network topology changes. This can be implemented leveraging mainly the G operator as follows:

```
def channel(source, destination, width) =
  gradient(source) + gradient(destination)
    <= distanceBetween(source, destination) + width

def distanceBetween(source, destination) =
  broadcast(source, gradient(destination))
```

Self-Organising Coordination Regions (SCR) In a more advanced scenario, one could employ the SCR design pattern. The core objective of the SCR pattern is to partition a decentralized system into distinct spatial zones, each overseen by a designated leader device. This leadership device is responsible for consolidating data from other devices within its jurisdiction, and subsequently disseminating decisions that enforce policies across the entire region.

For example, in a large-scale project aimed at monitoring and controlling temperature, the SCR pattern allows the formation of uniformly sized regions. Within these zones, devices collaboratively compute the area's average temperature. Using this aggregated information, the system could trigger alarms, facilitating more effective, coarse-grained analysis and intervention measures. This pattern is implemented using the G, C and S operators as follows:

```
val leader = S(radius, nbrRange)
val potential = G(leader, 0.0, distance => distance + nbrRange)
val totalTemperature = C(potential, _ + _, temperature(), 0)
val nodeInArea = C(potential, _ + _, 1, 0)
val averageTemperature = totalTemperature / nodeInArea
val decision = ....
val leaderDecision = broadcast(leader, decision)
```

3.3.3 Tools

Proper software tooling is essential to new self-organising algorithms and variants or extensions of the aggregate programming model, promoting scientific and technological progress. In the following, we present ScaFi–a Scala-based aggregate programming framework–and Alchemist–a meta-simulator for aggregate computing.

3.3.3.1 ScaFi

ScaFi (Scala-Fields) is an aggregate programming toolkit that comprises an internal DSL (language and virtual machine) as well as supporting components for the simulation and execution of aggregate systems.

Software description ScaFi is a multi-module Scala project hosted on GitHub[2]. It provides DSL and API modules for writing, testing, and running aggregate programs, namely programs expressed according to the aggregate programming paradigm [BPV15; Vir+19].

Software Architecture The high-level architecture of ScaFi is depicted in Figure 3.7. It consists of the following main components:

- `scafi-commons` — provides basic abstractions and utilities (e.g., spatial and temporal abstractions);
- `scafi-core` — provides an aggregate programming DSL (syntax, semantics, and a virtual machine for evaluation of programs), together with a "standard library" of reusable functions;
- `scafi-stdlib-ext` — provides extra library functionality that requires external dependencies and is hence kept separated from the minimalist `scafi-core`;
- `scafi-simulator`: provides basic support for simulating aggregate systems;
- `scafi-simulator-gui` — provides a GUI for visualizing and interacting with simulations of aggregate systems;
- `spala` ("spatial Scala"—i.e., a general aggregate computing platform[3]) — provides an actor-based aggregate computing middleware (independent of the ScaFi DSL and potentially applicable to other aggregate programming languages as well) based on the Akka toolkit [RBW15];
- `scafi-distributed` — ScaFi integration-layer for `spala`, which can be leveraged to set up actor-based deployments of ScaFi-programmed systems.

ScaFi leverages the concept of an *incarnation*, namely a concrete "family of types" [OZ05] that is progressively refined through inheritance, composed, and finally instantiated into an object (cf. the Scala *cake pattern* [Hun13; OZ05]) which ultimately provides access to a type-coherent set of features.

Figure 3.8 provides an excerpt of the main Scala traits with some types and objects they define. Trait `Core` provides the abstract fundamental types: `CNAME` for capability names, `ID` for device identifiers, `Context` for the input environment of computation rounds, and `Export` for the outcomes of computation rounds. Trait `Language` provides the syntax of the DSL in terms of methods, through

[2] `https://github.com/scafi/scafi`

[3] aggregate computing is rooted in spatial computing [Bea+12].

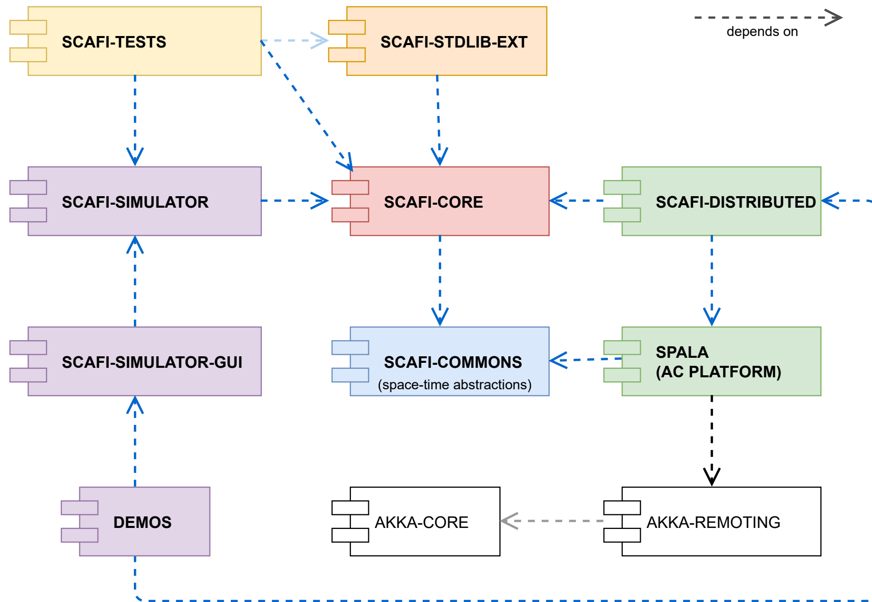


Figure 3.7: High-level architecture of the ScaFi toolkit.

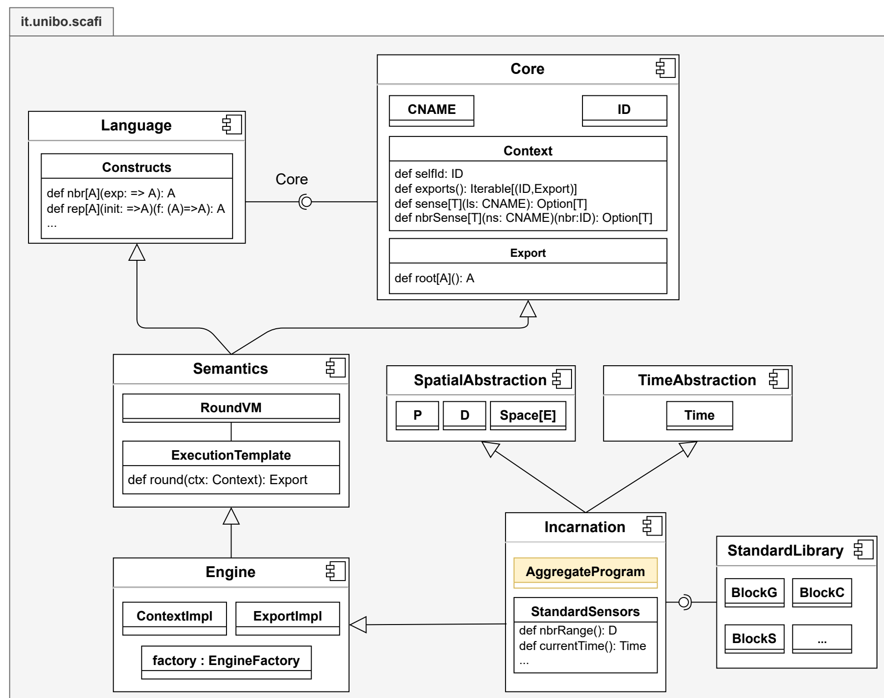


Figure 3.8: Design of the core of ScaFi (DSL).

interface `Constructs`. Trait `Semantics` and `Engine` implement the DSL construct semantics, providing a template for `AggregateProgram` base class defined in the `Incarnation` trait. The incarnation also exposes `StandardSensors` in terms of, e.g., `SpatialAbstraction`'s and `TimeAbstraction`'s types for positions (`P`), distances (`P`), and time. The `StandardLibrary` is provided by leveraging what an incarnation provides, providing traits of functionality to be mixed into `AggregatePrograms`.

Software Functionalities

Expressing aggregate programs through a Scala DSL Module `scafi-core` exposes, through incarnations, an `AggregateProgram` trait that provides access to aggregate programming constructs—following a variant of the field calculus [Aud+19b; Vir+19] formalized in [Cas+20a]. This single program defines – from a global perspective – the collective adaptive behaviour of an entire ensemble of computational devices. Besides the core constructs, this module also provides "standard library" traits providing access to reusable functions of aggregate functionality. For instance, by mixing trait `Gradients` into an `AggregateProgram` subclass, a developer gets access to *gradient functions* [Bea+08; Vir+18a], used to continuously compute (over space and time) the self-healing field of minimum distances of each node from a set of source nodes. Several such traits are available to provide other key building blocks for self-organising applications [WH07; Vir+18a] (e.g., `BlockG` for gradient-wise information propagation, `BlockC` for gradient-wise information collection, `BlockS` for sparse choice or leader election) or experimental language features (e.g., the `spawn` function for concurrent aggregate processes [Cas+21a; Tes+22], for modelling independent and overlapping aggregate computations).

Virtual machine for the local execution of aggregate programs An `AggregateProgram` instance is a function mapping a `Context` (the set of inputs needed by an individual device to properly evaluate the program locally) to an `Export` (the tree of values that has to be shared with neighbours to effectively coordinate and promote the emergence of collective behaviours). Using this API, a developer can integrate "aggregate functionality" into its system—what remains to be specified are the details of the aggregate execution model and the communication among devices, that may change in different applications. Devices must continuously run the aggregate program, but the scheduling of these computation rounds can be tuned as the application needs [Pia+21]. `Exports` must be shared with neighbouring devices to allow them to properly set up their `Contexts`, but the network protocol to be used to do so can be selected independently of the

program.

Simulation support In order to simulate an "aggregate system", it is necessary to:

1. define the set of computational devices that make up the aggregate, including their sensors and actuators;
2. define the aggregate topology, i.e., some application-specific *neighbouring relationship* from which the set of *neighbours* of each device can be determined;
3. define the aggregate program to be executed;
4. define a certain dynamics of the system by proper scheduling of computation rounds, and the environment by proper scheduling of changes in sensor values.

Module `scafi-simulator` provides this basic support. It exposes some factory methods to configure simulations properly (e.g., it supports ad-hoc and spatial distance-based connectivity rules) and an API to run and interact with simulations. Then, module `scafi-simulator-gui` provides a convenient graphical user interface to launch and visually show simulations in execution. We remark that these modules currently support basic simulation scenarios and are mainly meant for quick experiments or as a starting basis for ad-hoc simulation frameworks.

Experimental or work-in-progress features: actor-based middleware Regarding the construction of actual systems, ScaFi provides an actor-based implementation of the aggregate execution model [CV18], in the `spala` (`Spa`tial Sca`la`) module, which is instrumental for integrating aggregate computing into existing systems and distributed architectures [CV18]. Indeed, aggregate computing systems can be designed, deployed, and executed according to different architectural styles and concrete architectures [Cas+20b]. So, ScaFi provides *two* main implementations of the middleware, in package `it.unibo.scafi.distrib.actor`, for purely peer-to-peer (sub-package `p2p`) and server-based designs (sub-package `server`). The main abstraction is the `DeviceActor`, which exposes a message-based interface for controlling and interacting with an individual logical node of the aggregate system. Then, an object-oriented façade API is provided to set up a system of middleware-level actors.

Features ScaFi has been used in aggregate computing-related research [Cas+21a; Aud+22; ACV22b; CV19; Cas+19; CAV18; CAV21a; Cas+21b; Cas+22; Aud+20], touching themes such as software engineering,

computational models, and distributed systems/algorithms. The impact of ScaFi can be understood in terms of existing and prospective contributions, discussed in the following.

Interplay between programming language design and foundational research The implementation of the ScaFi DSL has inspired a variant of the field calculus which arguably supports easier embeddability into mainstream programming languages [Cas+20a; Aud+20].

High-level programming models The previous discussion makes the case for "DSL stacking" [HE10]. Indeed, by leveraging the aforementioned aggregate process extension, it is possible to reduce the abstraction gap needed to implement *situated tuples* [Cas+21b], which is a Linda-like model [Gel85] for coordinating processes where tuples and tuple operations are situated in space. By mapping high-level specifications into aggregate programs, it is sometimes straightforward to develop resilient distributed implementations—as in [Aud+21], where translation rules from spatial logic formulas to field calculus expressions enable seamless construction of decentralized monitors for such formulas.

Web-friendliness By leveraging Scala.js [Doe18], ScaFi can be easily accessed through JavaScript, which promotes cross-platform language design and reuse of functionality in the browser (to support web applications without the need of server-side components). This paved the path to ScaFi-Web [Agu+21], a web playground for aggregate programming.

3.3.3.2 Alchemist

Alchemist[4] [PMV13] is meta-simulator mainly designed for simulating complex distributed systems in a rich variety of scenarios like swarm robotics [ACV23], large-scale sensor networks [Agu+22], crowd simulation [BPV15], path planning, and even morphogenesis of multi-cellular systems.

The simulator is *meta* in nature, as it is based on general abstractions that can be mapped to specific use cases (i.e., *incarnations*). Inspired by biochemistry, the meta-model consists of a set of *nodes* that exist in an *environment* and are linked together by *relationship* rules. Each node contains a sequence of *molecules* and *reactions*. A *molecule* represents a variable, which acts as a container for data. *Reactions* instead are events that occur based on a set of *conditions*, and are fired according to a time distribution, producing an effect that is described as an action. This abstraction allows the simulator to be flexible and adaptable to

[4] `http://alchemistsimulator.github.io/`

```
incarnation: scafi
network-model:
  type: ConnectWithinDistance
  parameters: [0.5]
deployments:
  type: Grid
  parameters: [-5,-5,5,5,0.25,0.25]
  /*dynamics of the simulation*/
  programs:
    - program:
    - time-distribution: 1
      type: Event
      actions:
      - type: RunScafiProgram
       parameters: [program]
    - program: send
```

Listing (3.1) An Alchemist simulation example.

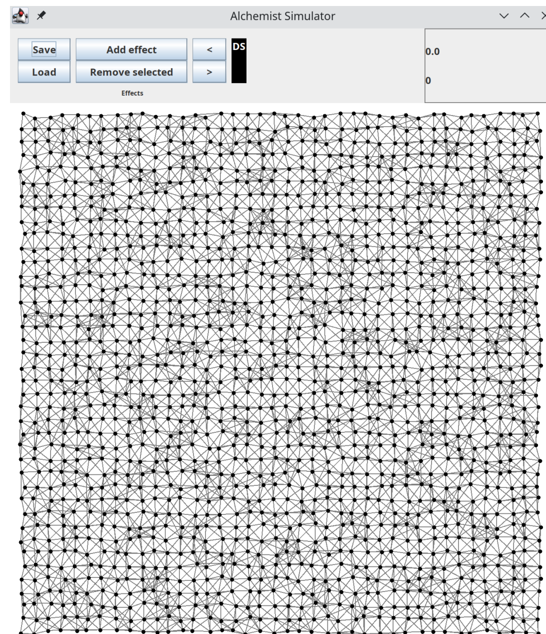


Figure 3.9: An Alchemist simulation example. The simulation result on the right is obtained by running the simulation described on the left.

a variety of use cases and node numbers (it could support thousands of nodes), while maintaining a consistent underlying structure.

The Alchemist simulator features four incarnations: biochemistry, Sapere, Protelis, and ScaFi, each with a different way of modelling molecules and actions. The latter is the reference of this thesis since it supports the ScaFi Scala DSL the current reference framework for aggregate computing in Scala.

Alchemist offers an effortless method for loading simulations. The process requires a YAML file that includes essential parameters, such as the incarnation type, neighbour connection model, and node deployment. In Figure 3.9, we have provided an example YAML file that creates a simulation using the ScaFi incarnation (first row). It also defines the neighbourhood relationship based on fixed distances (0.5 in this case), placing nodes in a fixed grid of size 10x10 starting at -(5,5) and ending at (5,5), with a node-to-node distance of 0.25. Finally, it loads the ScaFi program called “program”, which is evaluated at each node with a frequency of 1.

3.4 Final Remarks

Addressing collective adaptive behaviour is a long-standing research challenge in the realm of complex systems. Macro programming has emerged as a promising approach to bridge the abstraction gap between the problem space and the solution space.

In this chapter, we provide an overview of the key concepts within this programming paradigm and introduce aggregate computing—a specialized macro programming technique for orchestrating collective self-organizing behaviours in highly scalable and distributed systems. While this paradigm has already found applications in the field of CPSW, we identify several areas for further improvement:

- A high-level interface tailored for effectively programming behaviours specific to the CPSW domain, such as movement, pattern formation, and collective decision-making.
- Appropriate abstractions to facilitate the design and development of intricate CPSW systems.
- Innovative algorithms to enable complex collective behaviours, like sensing-driven clustering.
- Robust deployment strategies that accommodate modern, complex IT architectures.
- The incorporation of machine learning techniques to overcome current limitations in the state-of-the-art foundational frameworks.

Chapter 4

Reinforcement Learning

What is reinforcement learning?
What are the main models to solve reinforcement learning problems?
Can reinforcement learning be applied in many-agent systems?
What are the taxonomies and the main algorithms?
– **RQ1**

Contents

4.1 Single-agent 50
4.1.1 Markov Decision Process 51
4.1.2 Find a policy given an MDP 52
4.1.3 Find a policy without an MDP 55
4.1.4 Policy Gradient Methods 57
4.1.5 Approximate Solutions 59
4.1.6 Wrap up 60
4.2 Multi-agent 61
4.2.1 Stochastic games 62
4.2.2 Taxonomies 64
4.2.3 Solutions for MARL 67
4.2.4 Wrap up 71
4.3 Many-agent 71
4.3.1 Formalization 71

4.3.2 Solutions for many-agent reinforcement learning (ManyRL) 72
4.4 Final Remarks 76

The concept of intelligence is as complex as it is intriguing, and it has been a subject of philosophical inquiry, scientific exploration, and cultural curiosity for centuries. Philosophers have debated on what constitutes intelligence, linking it to *reason*, *wisdom*, and even *morality*. Despite these varied interpretations, defining intelligence remains a challenge, even in the field of psychology. Several standardized tests and scales attempt to measure intelligence, like the one developed by Alan Turing [Tur50], but none manage to capture the complete essence of what it means to be "intelligent". Intelligence is often understood as the ability to *learn*, *reason*, and *adapt*, among other cognitive abilities.

Among the myriad of perspectives on intelligence, learning stands out as a *fundamental* component. From an evolutionary point of view, the ability to learn is essential for survival. An organism that can adapt to its environment and learn from experiences is likely to survive and reproduce. In the human context, learning has been the cornerstone of development, be it mastering a language, solving complex problems, or creating art.

This notion of learning is crucial in the realm of Artificial Intelligence (AI). If intelligence involves learning, then replicating intelligence *artificially* would necessarily entail enabling machines to learn. This hypothesis leads us to the exciting and rapidly evolving domain of *machine learning*— a subset of AI that allows computers to learn from data, rather than requiring them to be explicitly programmed for specific tasks.

Machine learning encompasses various approaches and techniques that aim to make machines learn. Broadly, these approaches can be categorized into supervised, unsupervised, semi-supervised, and reinforcement learning. Particularly, the first three approaches are based on the idea of *learning from data*, where this data can be labelled or unlabelled:

- *supervised learning*: this is the most straightforward approach, where a model is trained on a labelled dataset. The model makes predictions or decisions based on input data, and it is corrected when its predictions are incorrect. Typical examples include classification and regression problems;
- *unsupervised learning*: unlike supervised learning, this approach does not involve labelled data. The machine tries to learn the patterns and the structure from the data without any supervision (e.g., clustering algorithms);
- *semi-supervised*: A middle-ground between supervised and unsupervised learning, this approach utilizes both labelled and unlabelled data for training.

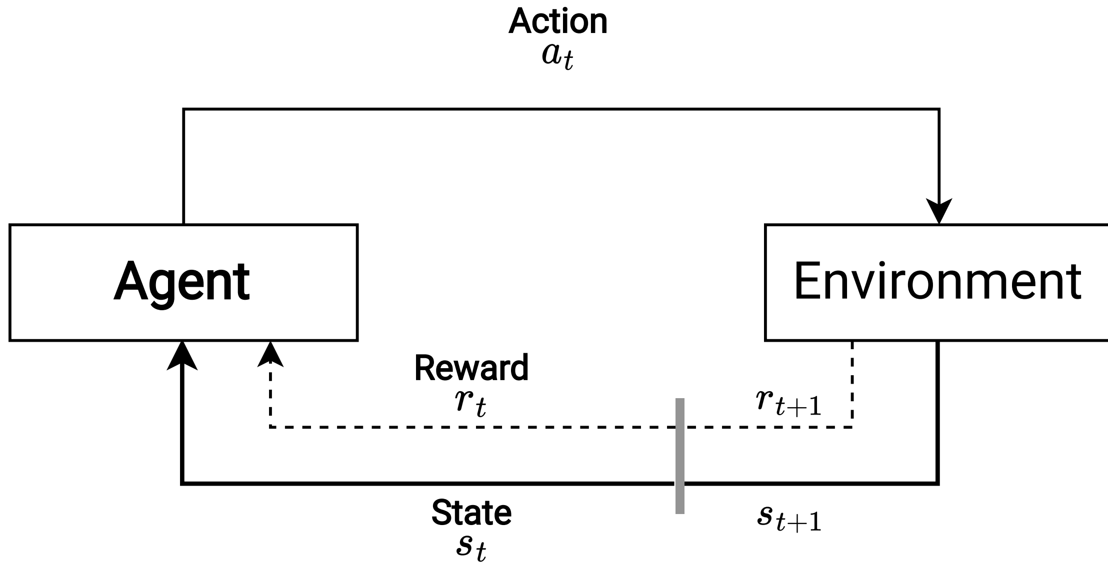


Figure 4.1: Overview of the reinforcement learning (RL) framework.

The model learns to improve its predictions gradually.

Reinforcement learning (RL) sets itself apart from other methodologies by operating without the need for labelled data or supervision and through a sequential interaction between an agent and an environment employing a *trial-and-error* strategy. Subsequent sections will delve into the nuances of this distinctive approach, starting from single-agent settings and then moving to multi-agent and many-agent systems.

4.1 Single-agent

Reinforcement learning [SB18] serves as a universal framework that has been inspired by the cognitive processes underlying human learning. This paradigm has proven to be highly effective for addressing *control problems*, which are essentially tasks that require decision-making to achieve a particular outcome. The core focus of RL is on the *sequential* interactions that occur between an *agent* and an *environment* (summarized in Figure 4.3). Agents are defined as entities capable of performing *actions*, while the environment constitutes everything external to the agents and beyond their immediate control.

During each discrete time step, denoted as t, an agent observes the current state of the environment s_t (e.g., the robot position according to a GPS sensor). This state encapsulates the set of all observable information at that particular moment.

The agent then proceeds to select an *action* a_t (e.g., the torque to be applied to engines) in accordance with its *policy* π. A policy serves as a probabilistic mapping that guides the agent in choosing actions based on the current state. Policies can be simple lookup tables or complex neural networks. As a result of taking this action, the environment transitions to a new state s_{t+1} at the next time step $t+1$. Simultaneously, the agent receives a *reward* r_{t+1}, which is a quantitative measure of the efficacy of the action taken, given the state of the environment.

The overarching objective of RL is to discover an *optimal* policy, denoted as π^*, that aims to maximize the long-term return, or cumulative reward, G. This is generally achieved through a *trial-and-error* learning process, where agents continually adapt their policies based on the rewards received.

This framework has found extensive applications in a diverse array of domains. For example, RL has been used to create advanced algorithms for video games [Aru+17], allowing for AI agents that can outperform human players. In robotics [KBP13], RL algorithms are enabling machines to learn complex tasks autonomously, from simple object manipulation to navigation in unstructured environments. It is also making significant inroads in networking, particularly in routing algorithms where dynamic decision-making is crucial [Luo+19].

4.1.1 Markov Decision Process

This general framework is supported by Markov Decision Process (MDP) [How60]— a mathematical model that describes the environment evolution in sequential decision problems. A MDP consists of a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} >$ in which:

- $\mathcal{S}$ denotes the set of states;
- $\mathcal{A}$ is the set of actions;
- $\mathcal{P}(s_{t+1}|s_t, a_t)$ define the probability to reach some state s_{t+1} starting from s_t and performing a_t (i.e. transition probability function);
- $\mathcal{R}(s_t, a_t, s_{t+1})$ devise a probabilistic reward function.

In MDP, $\mathcal{P}$ is *memory-less*, namely the next environment state depends only on the current state— that is the *Markov property*. Another important concept in MDP is the return G defined as the discounted sum of reward a possible future trajectory τ (i.e. a sequence of time steps):

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^T r_{t+T} = \sum_{k=t}^{T} \gamma^{k-t} r_k \tag{4.1}$$

Where $0 \leq \gamma \leq 1$ is the *discount factor*, that is how much the future reward impacts the long-term return. Based on the value of T, we can distinguish between *episodic* and *continuous* tasks. The foster ones are characterized by a finite number of time steps (e.g., a match of chess), while the latter ones are infinite (e.g., a robot that should wander in an unknown environment).

Reinforcement learning goal

The RL goal can be expressed as the maximization of the *expected* long-term return following a policy π:

$$J = \mathbb{E}_\pi\Big[G_t\Big] = \mathbb{E}_\pi\Big[\sum_{k=t}^{T} \gamma^{t-k} r_k\Big] \tag{4.2}$$

Particularly, in RL we want to find the optimal policy π^* that maximizes J:

$$\pi^* = \arg\max_\pi J \tag{4.3}$$

The equation essentially captures the trade-off between immediate and future rewards. The agent aims to select actions based on the policy π that will maximize this expected long-term return. The discount factor γ allows us to model the agent's consideration for future rewards and is a hyperparameter that can be tuned based on the specific problem being solved.

4.1.2 Find a policy given an MDP

In the context of MDP, evaluating policies is crucial for identifying the optimal one. Consequently, two essential concepts are introduced: the *value function* and the *Q-function*. V^π is the value function that evaluates how good (or bad) a *state* is according to the long-term return following the policy π (*expected value*). It is defined as:

$$V(s)^\pi = \mathbb{E}_\pi\Big[G_t | s_t = s\Big] \tag{4.4}$$

Q^π is the corresponding value function that evaluates *state-action* pairs:

$$Q(s,a)^\pi = \mathbb{E}_\pi\Big[G_t | s_t = s, a_t = a\Big] \tag{4.5}$$

Policies could be defined through value functions. In particular, a greedy policy based on Q function is the one that always chooses the action with the highest value in a certain state:

$$\pi(s) = \arg\max_a (Q(s,a)) \tag{4.6}$$

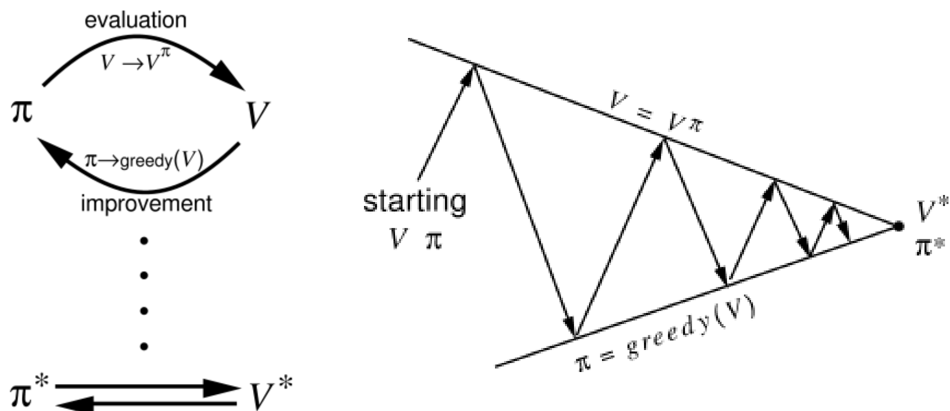


Figure 4.2: General schema of policy iteration (left) and value iteration (right) taken from [SB18].

4.1.2.1 Dynamic programming

Dynamic programming is a family of algorithms that can be used to compute optimal policies given a model of the environment as an MDP. In particular, the *Bellman equation* is a fundamental concept in dynamic programming. It is a recursive equation that decomposes the value function into two parts: the immediate reward obtained from the current state and the discounted value of the future state. The Bellman equation for the value function is defined as:

$$V(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) \Big[\mathcal{R}(s,a,s') + \gamma V(s') \Big] \tag{4.7}$$

Similarly, the Bellman equation for the Q function is defined as:

$$Q(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) \Big[\mathcal{R}(s,a,s') + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s') Q(s',a') \Big] \tag{4.8}$$

The Bellman equation is the basis for many algorithms that solve MDP, two most notable are *value iteration* and *policy iteration* (Figure 4.2).

4.1.2.2 Value iteration

Value iteration is an iterative algorithm used to compute the optimal value function V^* and, consequently, the optimal policy π^*. The algorithm is particularly useful when the state and action spaces are too large to solve directly through analytical methods. It is based on the principle of optimality, which states that if an optimal policy π^* exists, then it must satisfy the Bellman optimality equation.

The algorithm starts by initializing $V(s)$ for all states s to some arbitrary values, often zeros. It then iteratively updates the value of each state s using the Bellman optimality equation until the value function converges to V^*. The convergence is usually checked by measuring the difference between successive value functions and comparing it against a small threshold ϵ:

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) \Big[\mathcal{R}(s,a,s') + \gamma V_k(s') \Big] \tag{4.9}$$

The max operation ensures that the value function is updated to reflect the best possible action at each state. The term $\gamma V_k(s')$ represents the discounted future rewards, and $R(s,a,s')$ is the immediate reward. The transition probability $P(s'|s,a)$ models the uncertainty in the environment.

After the value function has converged to V^*, the optimal policy π^* can be extracted. The policy is determined by selecting the action that maximizes the expected return in each state, as given by:

$$\pi^*(s) = \arg\max_a \sum_{s' \in \mathcal{S}} P(s'|s,a) \Big[R(s,a,s') + \gamma V^*(s') \Big] \tag{4.10}$$

This policy is guaranteed to be optimal concerning the original MDP.

4.1.2.3 Policy Iteration

Policy iteration is another dynamic programming algorithm used for finding the optimal policy π^*. Unlike value iteration, policy iteration consists of two main steps: *policy evaluation* and *policy improvement*, which are repeated iteratively until the policy converges to π^*:

- policy evaluation: in this step, the value function V^π for the current policy π is computed until it stabilizes. The update rule is:

$$V_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} P(s'|s,a) \Big[R(s,a,s') + \gamma V_k(s') \Big] \tag{4.11}$$

- policy improvement: after evaluating V^π, the policy is updated to be greedy with respect to V^π:

$$\pi'(s) = \arg\max_a \sum_{s' \in \mathcal{S}} P(s'|s,a) \Big[R(s,a,s') + \gamma V^\pi(s') \Big] \tag{4.12}$$

The algorithm then returns to the policy evaluation step, using the new policy π', and continues until the policy no longer changes.

4.1.3 Find a policy without an MDP

While dynamic programming methods like value iteration and policy iteration offer powerful ways to find the optimal policy π^*, they come with a significant limitation: the need for a complete model of the environment. Specifically, these algorithms require knowledge of the transition probability function $\mathcal{P}$ and the reward function $\mathcal{R}$. In many real-world applications, these functions are either unknown or too complex to model accurately. This is where *model-free* algorithms come into play. In the following sections, we will discuss two such algorithms' families: Monte Carlo methods and Temporal Difference methods.

4.1.3.1 Monte Carlo methods

Monte Carlo methods offer a way to find an optimal policy π^* without requiring a model of the environment. These methods rely on sampling sequences of states, actions, and rewards from actual or simulated interactions with the environment. By averaging these samples, the agent can estimate the value functions $V(s)$ and $Q(s,a)$, which can then be used to improve the policy.

The core idea is to run multiple episodes, from start to finish, and then update the value estimates based on the returns observed. The value of a state s or a state-action pair (s,a) is estimated as the average of the returns that have followed that state or state-action pair across multiple episodes.

$$V(s) = \frac{1}{N}\sum_{i=1}^{N} G_t^{(i)} \tag{4.13}$$

$$Q(s,a) = \frac{1}{N}\sum_{i=1}^{N} G_t^{(i)} \tag{4.14}$$

Where N is the number of times the state or state-action pair has been visited, and $G_t^{(i)}$ is the return following the i-th visit.

Once the value functions are estimated, the policy can be improved by making it greedy concerning these estimated values. In this context, the *exploration-exploitation* trade-off is crucial. In fact, the agent should explore the environment to discover new states and actions that could lead to higher rewards, but it should also exploit the knowledge it has already acquired to maximize the expected return. Monte Carlo methods are particularly useful when the state and action spaces are large, making it impractical to enumerate all possible state-action pairs. However, they do require the episodes to be finite and can be computationally expensive due to the need for multiple samples to obtain accurate estimates.

The Exploration-Exploitation Dilemma in Reinforcement Learning The exploration-exploitation trade-off is a cornerstone challenge in reinforcement learning algorithms. An agent must *explore* its environment to uncover new states and actions that may yield higher rewards. Simultaneously, it needs to *exploit* its existing knowledge to maximize the expected return. While this challenge pervades many learning algorithms, it is especially prominent in model-free methods where the agent learns a policy without having a predefined model of the environment.

Various strategies exist to navigate this dilemma. One of the most prevalent is the ϵ-*greedy* policy. In this approach, the agent selects a random action with probability ϵ and the action that is currently estimated to be the best with probability $1 - \epsilon$. The ϵ value is generally initialized to a small constant, such as 0.1 or 0.2, to ensure a reasonable balance between exploration and exploitation. Mathematically, the policy is defined as follows:

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{|\mathcal{A}|} + 1 - \epsilon & \text{if } a = \arg\max_{a \in \mathcal{A}} Q(s, a) \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases} \tag{4.15}$$

A useful refinement of the ϵ-greedy strategy is to implement a *decaying* ϵ, which starts at a higher value and is reduced over time. This adaptive approach allows the agent to initially focus on exploration to a greater extent, then gradually shift toward exploiting its accumulated knowledge as its value function estimates stabilize.

4.1.3.2 Temporal difference methods

Temporal difference methods are a class of *model-free* and *value-based* algorithms that allow the agent to learn optimal behaviour directly from its interactions with the environment, without requiring a model. TD methods combine ideas from both Monte Carlo methods and dynamic programming to provide a flexible and powerful approach to reinforcement learning.

One of the simplest TD methods is *TD-Learning* [Sut88], which updates the value function $V(s)$ based on the temporal difference error δ, defined as $\delta = R(s, a, s') + \gamma V(s') - V(s)$. The value function is then updated using $V(s) \leftarrow V(s) + \alpha\delta$, where α is the learning rate.

Starting from this intuition two main approaches have been developed: *Q-learning* [WD92] and *SARSA* [SB18].

Q-Learning Q-Learning is an off-policy TD algorithm that focuses on learning the action-value function $Q(s, a)$. The update rule for Q-Learning is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \Big[R(s, a, s') + \gamma \max_{a'} Q(s', a') - Q(s, a) \Big] \tag{4.16}$$

The algorithm is termed "off-policy" because it learns the value of the optimal policy regardless of the policy being followed, thanks to the $\max_{a'}$ term in the update rule.

The Q-Learning algorithm is guaranteed to converge to the optimal action-value function $Q^*(s,a)$ as long as all state-action pairs are visited infinitely often and the learning rate α is sufficiently small. For the same reason of decay ϵ-greedy policy, it is also common to use a decaying learning rate, which starts at a higher value and is reduced over time. The full algorithm can be found in Algorithm 1.

Algorithm 1: Q-Learning

```
Initialise: Q(s, a) = 0 for all s ∈ S, a ∈ A
repeat
    for t = 0, 1, 2, ... do
        Observe current state s^t
        if With probability ϵ then
            Choose random action a^t ∈ A
        else
            Choose action a^t ∈ arg max_a Q(s^t, a)
        end
        Apply action a^t, observe reward r^t and next state s^{t+1}
        Q(s^t, a^t) ← Q(s^t, a^t) + α [r^t + γ max_a Q(s^{t+1}, a) − Q(s^t, a^t)]
    end
until for every episode
```

SARSA SARSA (State-Action-Reward-State-Action) is an on-policy TD algorithm. Unlike Q-Learning, SARSA takes into account the current policy π during the learning process. The update rule for SARSA is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[R(s,a,s') + \gamma Q(s',a') - Q(s,a)\Big] \tag{4.17}$$

where a' is the action taken under the current policy π.

Both Q-Learning and SARSA have their advantages and disadvantages. Q-Learning tends to find the optimal policy faster but may be more sensitive to noise. SARSA, being an on-policy method, is more conservative and tends to find safer policies, especially when the policy involves some level of risk or uncertainty.

4.1.4 Policy Gradient Methods

While temporal difference methods and Monte Carlo methods focus on learning value functions to derive optimal policies, *policy gradient* methods take a different

approach. They aim to directly optimize the policy π itself, rather than first estimating value functions. This is particularly useful in environments with high-dimensional action spaces, or continuous action spaces, where value-based methods may struggle. Moreover, in this way, it is possible to learn stochastic policies, which are often more robust and flexible than deterministic policies.

Policy gradient methods optimize the policy by ascending the gradient of the expected return concerning the policy parameters θ. Mathematically, this can be expressed as:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{4.18}$$

where $J(\theta)$ is the expected return when following policy π_θ, and α is the learning rate.

One of the foundational algorithms in this category is the REINFORCE algorithm [Sut+99]: it is one of the earliest and most straightforward policy gradient methods. It estimates the gradient of the expected return by sampling trajectories from the current policy π_θ. After each episode, the algorithm adjusts the policy parameters θ in the direction that increases the expected return. The core equation for the REINFORCE algorithm is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t \right] \tag{4.19}$$

Here, τ represents a trajectory sampled from the policy π_θ, and G_t is the return from time t. A trajectory, or episode, is defined as a sequence of states, actions, and rewards. The term $\nabla_\theta \log \pi_\theta(a_t|s_t)$ is the log-likelihood gradient, and G_t serves as a sample estimate for how good the action a_t is in state s_t.

The REINFORCE algorithm operates in an episodic setting, meaning it waits until the end of each episode to update the policy. This makes it well-suited for tasks where the episode termination is natural, such as games or tasks with a fixed time horizon. REINFORCE is particularly useful when the action space is high-dimensional or continuous, where traditional value-based methods like Q-Learning may struggle. However, one drawback of REINFORCE is that it can have high variance in its updates, which can make the training process unstable. Various techniques, such as using a *baseline* or employing advanced variance reduction methods, have been developed to mitigate this issue. Particularly, the *actor-critic* architecture is a popular approach that combines the advantages of both value-based and policy-based methods. In this paradigm, the policy is referred to as the *actor*, while the value function is referred to as the *critic*. The actor is responsible for selecting actions, while the critic evaluates the actions taken by the actor.

One popular extension of REINFORCE is proximity policy optimization (PPO) [Sch+17]—the algorithm used for training the ChatGPT model. PPO

is a policy gradient method that aims to improve the stability of the training process. It does so by limiting the size of the policy update at each iteration, which prevents the policy from changing too much between updates.

4.1.5 Approximate Solutions

While the fundamental algorithms discussed in previous sections provide a strong theoretical foundation for model-free RL, they often fall short in real-world applications. The primary challenges they face are:

- **Curse of dimensionality**: the state and action spaces in practical problems can be so large that enumerating them becomes computationally infeasible.
- **Partial observability**: in many scenarios, the agent cannot fully observe the entire state of the environment, making it difficult to make optimal decisions.

To illustrate the curse of dimensionality, consider the seemingly simple game of Go. The game's state space consists of 2^{170} possible states, a number so astronomical that it exceeds computational capabilities—especially when compared to the estimated 10^{80} atoms in the observable universe.

Similarly, partial observability is a pervasive issue in real-world applications. For example, a self-driving car perceives its environment through sensors, offering only a limited, partial view of the world. This restricted perspective can significantly impact the agent's ability to make optimal decisions.

Given these challenges, approximate solutions become not just desirable but often *necessary*. These solutions leverage function approximation techniques to estimate value functions or policies. While they may sacrifice some theoretical convergence guarantees, they offer a more practical approach to tackling complex, high-dimensional, and partially observable problems commonly encountered in real-world applications. In contemporary applications, neural networks have emerged as the go-to function approximators due to their exceptional capability to approximate complex, high-dimensional functions. Particularly, the combination of *deep learning* and reinforcement learning has led to significant advancements in the field, in the so-called area of *deep reinforcement learning*. One of the first and most influential works in this area was the Deep Q-Learning (DQL) algorithm [Mni+13], which will be discussed in the next section.

4.1.5.1 Deep Q-Learning

Deep Q-Learning represent a landmark innovation in the field of deep reinforcement learning, effectively combining the strengths of Q-Learning with the function

approximation capabilities of deep neural networks. DQL was initially designed to master a variety of Atari 2600 games and has since been adapted for various complex tasks.

The core idea behind DQL is to use a neural network as a function approximation for the Q-function in Q-Learning. The neural network, often referred to as the Q-network, takes the environment state as input and outputs Q-values for each action. Mathematically, the Q-network aims to approximate the optimal Q-function $Q^*(s,a)$ as closely as possible.

$$Q(s,a;\theta) \approx Q^*(s,a) \tag{4.20}$$

where θ represents the parameters of the neural network. However, directly applying neural networks to Q-Learning presents challenges, primarily due to the correlation between consecutive experiences and the non-stationary nature of the data. DQL addresses these issues through two key innovations:

- **Experience replay**: DQL stores past experiences (s,a,r,s') in a replay buffer and samples mini-batches randomly during training. This decorrelates the data and leads to more stable training.
- **Target network**: DQL introduces a separate, slowly updated target network to calculate the target Q-values, reducing the overestimation bias and improving stability.

The Q-value update equation in DQL is:

$$Q(s,a;\theta) \leftarrow Q(s,a;\theta) + \alpha \Big[r + \gamma \max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta) \Big] \tag{4.21}$$

Where θ^- are the parameters of the target network.

4.1.6 Wrap up

In conclusion, this section has provided a comprehensive overview of RL, beginning with the formulation of the problem and the underlying mathematical framework. We also explored various analytical solutions and key algorithms associated with RL.

Reinforcement learning algorithms can be categorized along multiple dimensions (summarized in Figure 4.3). One primary distinction is between *model-free* and *model-based* algorithms. In the former, there is no need for a model of the environment, allowing the algorithm to learn directly through interaction. In contrast, model-based algorithms require an MDP to compute the optimal policy. Another important categorization is whether an algorithm is *on-policy* or *off-policy*. On-policy algorithms optimize the same policy that is used for exploration, whereas

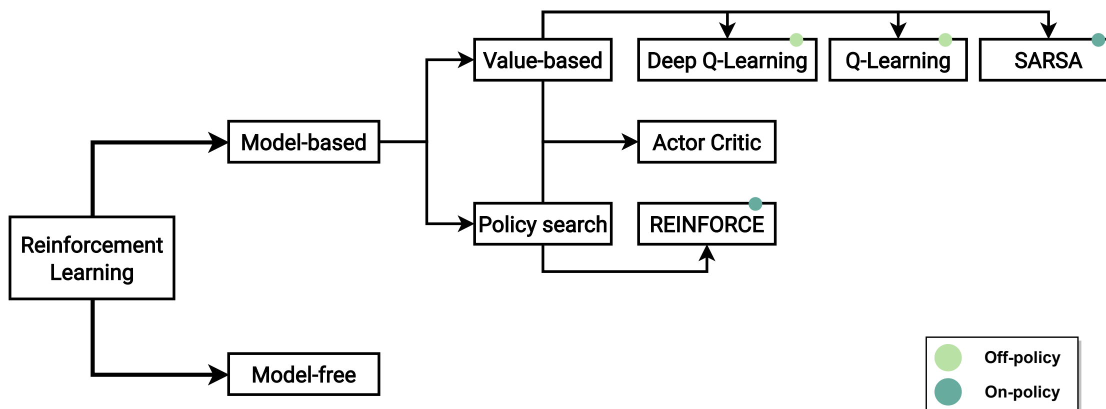


Figure 4.3: Overview of the RL algorithms.

off-policy algorithms utilize two separate policies during the learning process, commonly referred to as the behaviour and target policies. RL algorithms can be broadly divided based on what they aim to learn. The primary categories here are value-based and policy gradient methods, although hybrid approaches also exist that combine elements of both, like actor-critic. Lastly, RL algorithms can be categorized based on whether they use a *tabular* or *approximate* approach. Tabular methods store the value function in a table, while approximate methods leverage function approximation techniques, such as neural networks, to estimate the value function.

This overview serves as a basis for the multi-agent and many-agent RL algorithms, because most of the algorithms that we will discuss in the following sections are based on the concepts presented here.

4.2 Multi-agent

In the evolving landscape of RL, the concept of multi-agent reinforcement learning (MARL) [Tan93a; Gu+21; OH19] (Figure 4.4) stands as a natural extension of the foundational RL principles. While traditional RL generally focuses on the interactions between a single agent and an environment, MARL broadens the scope to include multiple agents, each with its own objectives, policies, and decision-making processes.

The basic setting in MARL comprises multiple agents interacting either *cooperatively*, *competitively*, or in a *mixed* fashion within a shared environment—more details in the following sections. Each agent i observe its the environment state s_t^i, select actions a_t^i according to its policy π^i, and receive rewards r_{t+1}^i. However, in MARL, an agent's actions can directly or indirectly influence the states and re-

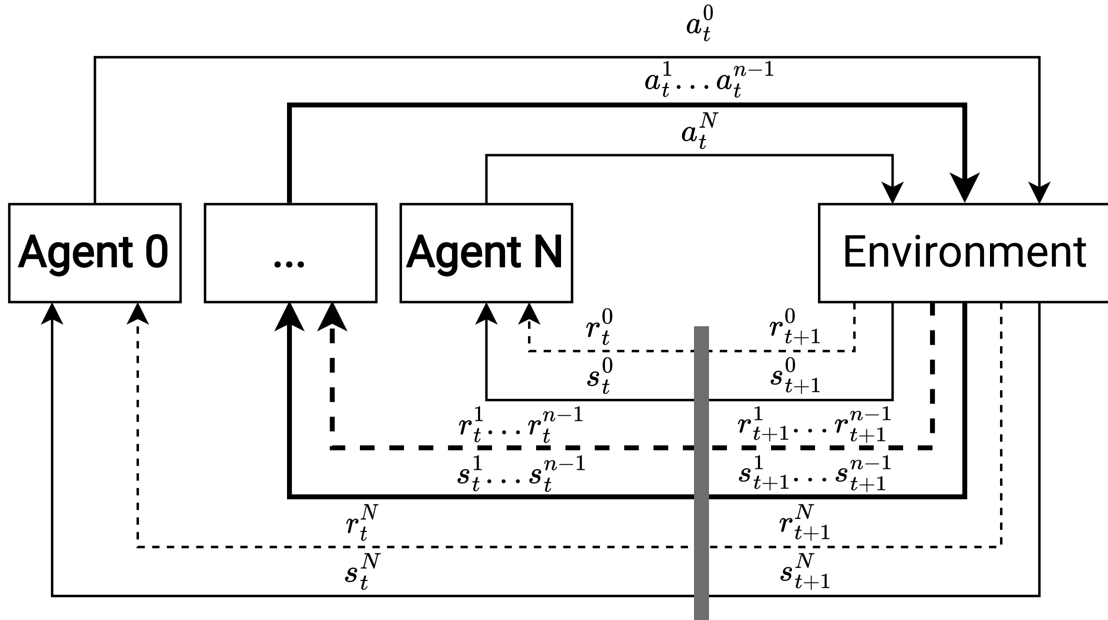


Figure 4.4: Overview of the MARL framework.

wards of other agents, thereby increasing the complexity of the learning problem. The key challenge in MARL is to develop robust algorithms that enable agents to learn optimal policies in these complex, often *non-stationary*, environments. Classic RL algorithms often require modifications to accommodate the multi-agent setting. For instance, the concept of a joint action space, a state space extended to multiple agents, and a composite reward function are essential considerations.

4.2.1 Stochastic games

The formalization of MARL typically extends the standard Markov Decision Process (MDP) framework to account for multiple agents. One of the most straightforward extensions is the Markov Game, also called Stochastic games [NS03]. In this formalization, each agent has its own state, action, and reward function, and the joint actions of all agents determine the transition dynamics and rewards:

$$S = \langle \mathcal{N}, \mathcal{S}, \mathcal{A}_1, \dots, \mathcal{A}_N, \mathcal{P}, \mathcal{R}_1, \dots, \mathcal{R}_N \rangle \tag{4.22}$$

Where:

- $\mathcal{N}$ is the number of agents. With $N = 1$ we have a single-agent setting, while with $N >> 2$ we have a many-agent setting.

- $\mathcal{S}$ is the environment state space. The environment is then considered to be fully observable.
- $\mathcal{A}_i$ is the action space of agent i. We denote the joint action space as $\mathbb{A} = \mathcal{A}_1 \times \ldots \times \mathcal{A}_N$.
- $\mathcal{P} : \mathcal{S} \times \mathbb{A} \rightarrow \Delta(\mathbb{S})$ is the joint transition probability function. For each time step t, $\mathcal{P}$ is a function of the joint action $a_t \in \mathbb{A}$ and the current state $s_t \in \mathcal{S}$, and returns the probability of transitioning to the next state $s_{t+1} \in \mathcal{S}$.
- $\mathcal{R}_i : \mathbb{S} \times \mathbb{A} \times \mathbb{S} \rightarrow \mathbb{R}$: is the reward function for agent i. The joint reward function is defined as $\mathbb{R} = \mathcal{R}_1 + \ldots + \mathcal{R}_N$.

The game can be described sequentially as follows:

1. At each time step t, each agent i observes the current state s_t^i.
2. Each agent i selects an action a_t^i according to its policy π^i.
3. The joint action $a_t = (a_t^1, \ldots, a_t^N)$ is executed, and the environment transitions to the next state s_{t+1} according to the transition probability function $\mathcal{P}$.
4. Each agent i receives a reward r_{t+1}^i according to the reward function $\mathcal{R}_i$.

It is important to note that in these games, actions from all agents are executed simultaneously, and each agent's reward is a function of the joint action taken by all agents. In many real-world scenarios, assuming the availability of a global system state is impractical. As such, an often-employed extension to the traditional Stochastic Game model is the *Partially Observable Stochastic Game* (POSG) [LW22]. This extension can be formally represented as follows:

$$\text{POSG} = \langle \mathcal{N}, \mathcal{S}, \mathcal{A}_1, \ldots, \mathcal{A}_N, \mathcal{P}, \mathcal{R}_1, \ldots, \mathcal{R}_N, \mathcal{O}_1, \ldots, \mathcal{O}_N, \mathcal{O} \rangle \tag{4.23}$$

Here, the primary divergence from the conventional Stochastic Game is the introduction of observation functions $\mathcal{O}_i : \mathcal{S} \rightarrow \mathcal{O}_i$, which maps the environment state to the observation space of agent i. Consequently, the agents' policies are now parameterized by their respective observation spaces $\mathcal{O}_i$ rather than the global state space $\mathcal{S}$. This alteration significantly influences both the algorithmic approaches and the complexities involved in solving POSGs.

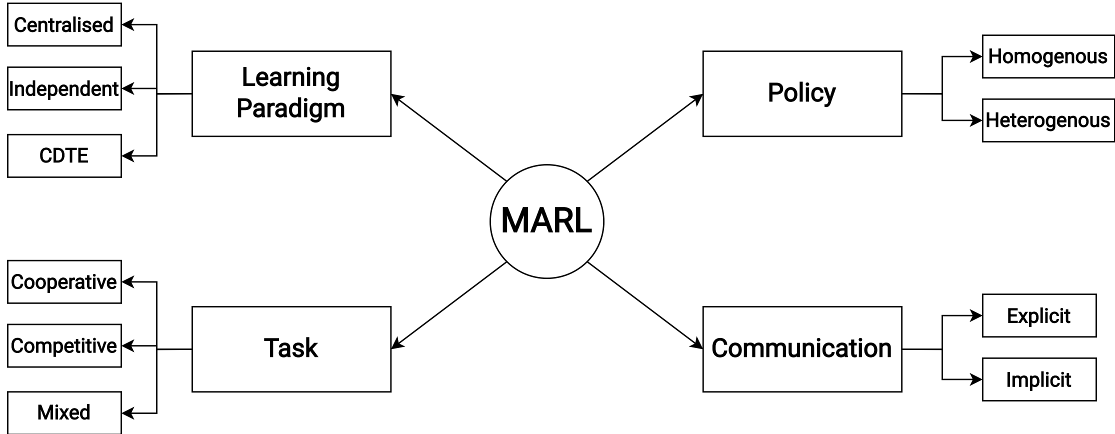


Figure 4.5: Overview of the MARL taxonomies.

4.2.2 Taxonomies

In single-agent settings, the focus of learning configurations is primarily on *algorithmic* aspects, such as whether an approach is value-based or relies on policy search, or whether it operates on-policy or off-policy. However, the landscape becomes more complex when multiple agents are involved. In such multi-agent settings, several dimensions must be considered to appropriately classify algorithms and approaches. Various surveys [Gu+21; OH19] have attempted to categorize the diverse classes of algorithms specifically designed for multi-agent scenarios. Each of these surveys emphasizes a unique characteristic or aspect of the problem under consideration. This section endeavours to outline various dimensions along which multi-agent algorithms can be categorized (Figure 4.5). The objective is to provide a framework for situating the algorithms that will be discussed in subsequent sections, particularly those that pertain to the *many-agent* perspective. Note that the following taxonomies are not mutually exclusive, and many algorithms can be classified along multiple dimensions.

4.2.2.1 Learning Paradigms

One possible categorization of multi-agent algorithms is based on the learning paradigm (summarized in Figure 4.6). In this context, the learning paradigm refers to how agents learn their policies. The field of multi-agent learning has historically been dominated by two primary approaches: *independent learning* and *centralized learning*. In independent learning, each agent learns its own policy concurrently and independently of the other agents in the environment [Tan93b]. In contrast, centralized learning involves a single learning entity that constructs a joint policy

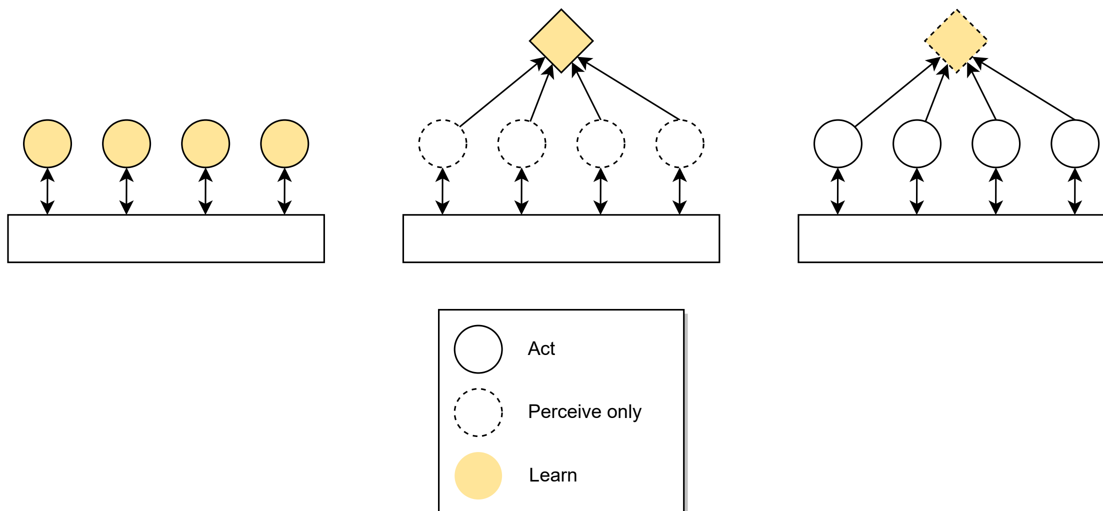


Figure 4.6: Overview of the MARL learning paradigms.

to control all agents.

Independent learning excels in scalability but often results in unstable learning dynamics due to the non-stationarity of the environment. This non-stationarity stems from the concurrent learning of multiple agents, making it challenging to learn effective policies. On the other hand, centralized learning benefits from environmental stability owing to the global view it maintains. However, it suffers from scalability issues due to the exponential growth of the state and action spaces as the number of agents increases. To illustrate, consider a system comprising N agents, each capable of taking M actions. In a centralized framework, the number of possible joint actions would be M^N. For instance, with $N = 100$ and $M = 2$, the number of joint actions would be an astronomical 2^{100}.

Recognizing the limitations of both independent and centralized learning paradigms, recent research has proposed a compromise: Centralized Training with Decentralized Execution (CTDE) [Low+17]. In this approach, agents are trained using a *global* view of the environment to learn a distributed policy. Indeed, during execution, each agent relies solely on its *local* observations to select actions. This hybrid method is particularly advantageous in partially observable environments: agents can learn a policy influenced by global states during the training phase while requiring only local observations for decision-making during execution.

4.2.2.2 Task type

Aside from how the agent learns, another important distinction involves *what* the agent should learn. The task type can be *cooperative*, *competitive*, or *mixed*.

Cooperative In cooperative tasks, agents are geared towards a common objective, aiming to maximize a shared reward. Formally, this is represented as a unified reward function for all agents, given by:

$$\mathcal{R}_0 = \mathcal{R}_1 = \ldots = \mathcal{R}_N = \mathcal{R} \tag{4.24}$$

Here, the objectives of all agents are perfectly aligned, necessitating coordination and collaboration among them to achieve the shared goal. Such cooperative scenarios are often encountered in the domain of CPSW, where inter-agent coordination is crucial for realizing collective outcomes effectively and efficiently.

Competitive In competitive tasks, agents operate under divergent objectives, and the reward function is usually tailored to each agent's performance. This can be formally expressed in two-player games as:

$$\mathcal{R}_0 = -\mathcal{R}_1 \tag{4.25}$$

In this framework, the agents have conflicting goals, creating a competitive landscape where each agent aims to maximize its reward.

Mixed In mixed tasks, agents operate under both cooperative and competitive dynamics. This is often encountered in real-world scenarios, where agents may have both shared and divergent objectives. This is the most general setting, and it is often the most challenging to solve.

4.2.2.3 Policy type

Another crucial aspect to explore is the nature of the policy that an agent ought to learn. Although numerous classifications of policies exist—ranging from deterministic to stochastic, centralized to decentralized, and joint to individual—this thesis places special focus on distinguishing between *homogeneous* and *heterogeneous* policy family.

Homogeneous Policies In the context of homogeneous policies, all agents *share* a common policy. The learning process aims to identify a *single*, *unified* policy applicable to all agents. This is particularly prevalent in cooperative tasks where agents must collaborate to achieve a collective objective. Utilizing a single policy across all agents often simplifies the learning process and enhances coordination among the agents. Moreover, is particularly suitable for scenarios where agents are *interchangeable* and *indistinguishable*, making it possible to use the same policy in different agent populations.

Heterogeneous Policies Conversely, in the realm of heterogeneous policies, each agent possesses its distinct policy. The learning algorithm is tasked with discovering a unique policy tailored to each agent. This approach, even if it can be also used in the context of cooperative tasks, is especially useful in competitive environments, where agents have disparate objectives and strive to optimize their rewards. Employing unique policies for each agent allows for the development of diverse strategies and behaviours, accommodating the complex dynamics of competitive settings.

4.2.2.4 Communication

Another important dimension to consider is whether agents are allowed to communicate with each other. In many real-world scenarios, agents are capable of exchanging information with other agents in the environment. This communication can be *explicit*, such as through the exchange of messages, or *implicit*, such as through the observation of other agents' actions.

In contrast, in other scenarios, agents are not allowed to communicate with each other. This is often the case in competitive settings, where agents are not permitted to share information. This restriction is often imposed to increase the complexity of the problem and to encourage the development of more sophisticated strategies.

4.2.3 Solutions for MARL

Addressing the intricacies and challenges inherent in MARL requires the design and implementation of sophisticated algorithms and frameworks capable of navigating a complex landscape shaped by multi-agent interactions, a diverse range of tasks, and varied learning paradigms. Over the years, a plethora of methodologies have emerged to confront these obstacles. Specifically, this thesis provides a comprehensive overview of approaches rooted in the RL domain.

A significant body of literature instead focuses on game-theoretical perspectives, which predominantly address scenarios involving a limited number of agents engaged in competitive tasks. Such approaches often incorporate concepts of *equilibrium* and *convergence* towards stable policies, typically framed within the context of Nash equilibrium. However, in practice, the pursuit of equilibrium often fails to yield satisfactory results, as it does not necessarily lead to agents learning optimal policies. Consequently, a substantial portion of the literature eschews equilibrium-centric views, opting instead to develop algorithms aimed at effective policy learning, even in the absence of formal proofs guaranteeing convergence to an equilibrium.

4.2.3.1 Tabular Methods

The first era of multi-agent reinforcement learning (MARL) algorithms primarily relied on tabular methods, based on variation of Q-Learning and SARSA. These methods are essentially extensions of single-agent tabular algorithms adapted for a multi-agent setting. In this section, we will briefly discuss some of the most popular tabular algorithms, laying the groundwork for the discussion of more advanced approaches in subsequent sections.

Distributed Q-Learning is a straightforward extension of the traditional Q-Learning algorithm designed for multi-agent systems. In this approach, each agent maintains its own Q-table and updates it based on its local observations and rewards. while this method allows for decentralized learning, it often struggles with issues related to *coordination* and *convergence* to global optima. The algorithm is particularly useful in scenarios where communication between agents is limited or not possible. However, it may not always guarantee convergence to the optimal joint policy, especially in complex environments where coordination between agents is crucial.

Hysteretic Q-Learning [MLF07] is a more recent addition to the family of tabular MARL algorithms. It addresses some limitations of Distributed Q-Learning, particularly in the context of cooperative multi-agent systems. In Hysteretic Q-Learning, each agent employs *two* learning rates for updating its Q-values, depending on whether the received reward is better or worse than expected. Formally, the update rule is:

$$\delta \leftarrow r - Q_i(a_i)$$

$$Q_i(a_i) \leftarrow \begin{cases} Q_i(a_i) + \alpha\delta & \text{if } \delta \geq 0 \\ Q_i(a_i) + \beta\delta & \text{else} \end{cases}$$

This dual learning rate mechanism allows for more flexible and efficient coordination among agents. Experimental results have shown that Hysteretic Q-Learning not only converges faster but also achieves comparable performance to centralized methods with significantly less computational overhead.

QD-Learning [KMP13] is a distributed version of the traditional Q-Learning algorithm tailored for multi-agent systems. Unlike centralized approaches, QD-Learning allows agents to engage in the learning process autonomously through *local* communication and computation. Each agent maintains a sequence of Q-matrices, and the Q-values for each state-action pair evolve in a collaborative

distributed manner. Specifically, the Q-values are updated according to a formula that incorporates both local one-stage costs and information from neighbouring agents.

The primary objective of QD-Learning is to minimize a network-averaged infinite horizon discounted cost. The algorithm achieves this by ensuring that each agent learns the value function based on locally accessible stochastic processes and one-stage cost processes. The agents collaborate using local processing and mutual information exchange, aiming to reach a consensus on the desired value function and the corresponding optimal control strategy. In terms of performance, QD-Learning has been shown to achieve optimal learning performance asymptotically, under minimal connectivity assumptions on the underlying communication graph. It has also been found to have a negligible asymptotic convergence rate loss compared to centralized Q-Learning making it a robust choice for distributed multi-agent systems. The update rule for QD-Learning is:

$$\begin{aligned} Q^n_{i,u}(t+1) = Q^n_{i,u}(t) - \beta_{i,u}(t) \sum_{l \in \Omega_n(t)} (Q^n_{i,u}(t) - Q^n_{l,u}(t)) \\ + \alpha_{i,u}(t) \left(c_n(x_t, u_t) + \gamma \min_{v \in U} Q^n_{x_{t+1},v}(t) - Q^n_{i,u}(t) \right) \end{aligned}$$

where $\beta_{i,u}(t)$ and $\alpha_{i,u}(t)$ are weight sequences adapted to the filtration $\mathcal{F}_n(t)$, and $c_n(x_t, u_t)$ is the local one-stage cost.

4.2.3.2 Approximate methods

The tabular methods discussed in the previous section are often impractical in real-world scenarios, as they require the storage of a Q-table for each agent. This can be computationally infeasible, especially in complex environments with large state and action spaces. To address this issue, a variety of approximate methods have been developed, leveraging function approximation techniques to estimate the Q-values or policies. The following are the most popular approximate methods for MARL, a complete overview however is out of the scope of this thesis and can be found in [OH19].

Counterfactual Multi-Agent (COMA) [Foe+17] is an actor-critic method specifically designed for cooperative multi-agent systems. Unlike traditional methods that rely on decentralized policies, COMA employs a centralized critic during the learning phase to estimate the Q-function. This centralized critic takes into account the global state or the joint action-observation histories, making it more effective in complex environments.

One of the key innovations in COMA is the introduction of a counterfactual baseline to address the challenge of multi-agent credit assignment. In cooperative

settings where joint actions produce global rewards, it becomes difficult for individual agents to understand their contribution to the team's success. The counterfactual baseline marginalizes a single agent's action while keeping the other agents' actions fixed, thereby providing a more accurate estimate of each agent's contribution. Another advantage of COMA is its computational efficiency. It employs a critic that allows the counterfactual baseline to be computed in a single forward pass, thereby reducing the computational burden.

COMA has been evaluated in complex scenarios like StarCraft unit micromanagement and has shown significant improvements over other multi-agent actor-critic methods. It is particularly useful in environments with high stochasticity, large state-action spaces, and delayed rewards.

Multi-Agent Proximal Policy Optimization (MAPPO) [Yu+21] is an extension of the widely-used Proximal Policy Optimization (PPO) algorithm, adapted for cooperative multi-agent systems. Unlike traditional methods that often rely on off-policy learning algorithms, MAPPO is an on-policy method that has shown to be surprisingly effective in multi-agent settings.

MAPPO employs a centralized value function during the training phase to estimate the Q-values. This centralized critic can take into account global state information, allowing MAPPO to follow a CTDE structure. This is particularly useful in complex environments where the state and action spaces are large, as it allows for more effective policy updates.

One of the standout features of MAPPO is its *sample efficiency*, that means the ability to learn from a few samples. Despite being an on-policy method, MAPPO is competitive or even superior to off-policy methods in terms of both final returns and sample efficiency. MAPPO also benefits from the robustness and stability of the underlying PPO algorithm. It requires minimal hyperparameter tuning and does not necessitate any domain-specific algorithmic modifications or architectures to achieve strong performance. This makes it a versatile and practical choice for a wide range of multi-agent environments.

QMIX [Ras+18] is a value-based method designed to address the challenges in multi-agent reinforcement learning (MARL).

It is based on the idea of *factorisation*, which involves decomposing the joint action-value function into a set of individual value functions. This factorisation allows for the extraction of decentralized policies that are consistent with the centralized training.

The architecture of QMIX consists of individual agent networks and a mixing network. Each agent network estimates an individual value function Q_a, while the mixing network combines these into a joint action-value function Q_{tot}. A

key feature of QMIX is the *monotonicity* constraint, which ensures that the joint action-value function is monotonic in the individual agent value functions. This allows for the extraction of decentralized policies that are consistent with the centralized training. QMIX is particularly effective in environments where the joint action spaces grow exponentially with the number of agents. It employs a mixing network with positive weights to enforce the monotonicity constraint, thereby ensuring that decentralized policies can be easily extracted. This makes QMIX computationally efficient and scalable, as it avoids the need for storing a Q-table for each agent.

4.2.4 Wrap up

In this section, we have provided an overview of MARL functional for this thesis perspective, beginning with the formulation of the problem and the underlying mathematical framework. We also explored various analytical solutions and key algorithms associated with MARL. In the following, we will discuss the *many-agent* perspective, understanding the challenges and the solutions proposed in the literature.

4.3 Many-agent

CPSW are characterized by a very large number of agents, with a population that can potentially range from hundreds to millions of agents. In such scenarios, the traditional MARL framework is no longer applicable, as the number of agents makes it computationally infeasible to learn a joint policy. A novel approach is required to address the challenges of many-agent systems, which is the focus of this section–in the so-called area of ManyRL [Yan21]. When we discuss many-agent systems, we refer to a large number of agents ($N \gg 2$) that are *interchangeable* and *indistinguishable*. In CPSW perspective, it is still applicable because we consider the agent to be behavioural homogeneous, even if they are not identical.

After this brief introduction, we will discuss the peculiarities of many-agent systems, starting from the formalization of the problem, and then we will discuss practical solutions in this context.

4.3.1 Formalization

Standard stochastic games do not capture the essence of many-agent systems, as they do not consider the homogeneous nature of the agents, and the large number of agents. Therefore, in the context of swarm robotics, a novel framework called *SwarMDP* [Šoš+16] has been proposed. This is focused on the idea of having a

swarming agent that rules the entire population of agents. In the following, we will discuss the formalization of this framework. Moreover, this model can be also applied in the system dynamics of aggregate computing discussed in Section 3.3.1.

4.3.1.1 SwarMDP

A SwarMDP is characterized by a *swarming agent* ($\mathbb{A}$) and the dynamics of the environment ($\mathbb{E}$). Specifically, $\mathbb{A}$ is a tuple $(\mathcal{S}, \mathcal{O}, \mathcal{A}, \mathcal{R}, \pi)$ where:

- $\mathcal{S}, \mathcal{O}, \mathcal{A}$ are the set of local states, observations (or features), and actions, respectively;
- $\mathcal{R} : \mathcal{S} \to \mathbb{R}$ is the reward function, which is influenced by the environment;
- $\pi : \mathcal{O} \to \mathcal{A}$ is the policy function, which maps the observations to the actions: it could be deterministic or stochastic.

Starting from this definition, the environment $\mathbb{E}$ is defined as a tuple $(\mathcal{P}, \mathbb{A}, \mathcal{T}, \xi)$, where:

- $\mathcal{P}$ is the total number of agents in the systems (the agent population), which is assumed to be fixed;
- $\mathbb{A}$ is the defined agent prototype that rules each agent $v \in P$;
- $\mathcal{T} : \mathcal{S}^P \times \mathcal{A}^P \times \mathcal{S}^P \to \mathbb{R}$ is the transition global function, which is influenced by the actions of the agents and returns a collective reward – this is typically not known by the swarming agents;
- $\xi : \mathcal{S}^\mathcal{P} \to \mathcal{O}^\mathcal{P}$ is the global observation model of the systems.

4.3.2 Solutions for ManyRL

In a ManyRL scenario, the concept of *homogeneity* is leveraged to design algorithms that scale effectively with the number of agents. Rather than developing N distinct policies for N agents, the objective is to formulate a *single* policy applicable to all agents. The learning paradigm typically employed in this context is CDTE, for the following reasons:

- fully decentralized learning is impractical due to the high level of non-stationarity that arises from concurrent learning;
- an agent's view of the system is so limited that locally collected experiences are insufficient for capturing the full range of possible local behaviour.

While several CDTE algorithms, such as MAPPO and QMIX, aim to find N policies, various practical adjustments can be employed to adapt them for multi-agent settings.

Subsequently, we will discuss the most popular of these techniques, as well as algorithms specifically designed for many-agent systems.

Parameter sharing

Parameter sharing [ACS23] is a pivotal technique in ManyRL, due to the strongly homogeneous agent assumptions. The essence of this approach lies in the utilization of a unified set of parameter values, denoted as θ_{shared} for the policy network or ϕ_{shared} for the value network, across all agents. This can be extended also to the case of standard RL algorithms, where the parameter sharing is applied to the Q-table. This unified parameter set serves as the backbone for both the policy and value functions of each agent. This learning approach is effective in many-agent systems, and it has several advantages:

- **Sample efficiency**: one of the most compelling advantages is the improvement in sample efficiency. By learning a single set of parameters, the algorithm can generalize across agents, thereby utilizing a more diverse and larger set of trajectories for training.
- **Computational efficiency**: parameter sharing keeps the computational load manageable by maintaining a constant number of parameters, irrespective of the number of agents. This is in stark contrast to a non-shared approach, where the number of parameters would increase linearly with the number of agents.

This, however, comes at the cost of policy space limitation: employing parameter sharing inherently constrains the policy space to identical policies for each agent, reducing the complexity of the overall joint policy. Said that, in CPSW this is quite reasonable, because complexity comes from the interaction of simple agents, rather than from complex agents.

Mathematically it involves constraining the policy parameters θ and/or value function parameters ϕ across agents as follows:

$$\theta_{\text{shared}} \equiv \theta_1 \equiv \theta_2 \equiv \ldots \equiv \theta_N$$

$$\phi_{\text{shared}} \equiv \phi_1 \equiv \phi_2 \equiv \ldots \equiv \phi_N$$

where N is the number of agents.

Experience Sharing

Experience sharing serves as another cornerstone in ManyRL [ACS23]. Unlike parameter sharing, which centralizes the neural network parameters across agents, experience sharing focuses on the distribution of experiences or trajectories among agents. By pooling experiences from multiple agents into a shared replay buffer, each agent has the opportunity to learn from a much richer set of experiences than it could generate on its own. This not only speeds up the learning process but also ensures a more uniform learning progression across agents, which is particularly useful for tasks that require coordinated multi-agent actions.

However, experience sharing comes with its own set of challenges. For instance, the approach can be computationally intensive as it increases the size of the batch used for training. Additionally, unlike parameter sharing, experience sharing allows for the possibility of agents learning diverse policies. However, these approaches can be combined to leverage the benefits of both techniques.

In terms of mechanics, experience sharing typically involves storing each agent's experiences, usually denoted as (s, a, r, s'), in a shared replay buffer. During the learning phase, agents sample from this shared buffer to update their individual policies and value functions.

Many-agent Q-Learning

Many-agent Q-Learning extends the classic Q-Learning algorithm to a multi-agent setting by incorporating the concepts of experience sharing and parameter sharing. In this approach, each agent maintains a global Q-table but can learn from a shared experience replay buffer. This shared buffer contains the state-action-reward-next state tuples (s, a, r, s') from all agents, thereby enriching the learning process for each individual agent.

The algorithm operates similarly to traditional Q-Learning, with the primary difference being the source of experiences used for updates. During each learning iteration, agents sample experiences from the shared replay buffer to update the global Q-values. The Q-value update equation remains the same as in single-agent Q-Learning, but the experiences used for the updates are drawn from the collective experiences of all agents.

By leveraging shared experiences, this approach aims to achieve faster convergence and more robust policies. Agents benefit from the diverse set of experiences, which can be particularly useful in complex or stochastic environments where individual experiences may not provide sufficient coverage of the state-action space.

This method is especially useful in scenarios where agents are working towards a common goal but partial observability of the environment. By sharing experiences, they can collectively learn a more comprehensive and effective strategy to achieve

their objective. This can be also extended to deep-q learning approaches, where the Q-table is replaced by a neural network. The algorithm is summarized in

Algorithm 2: Many-agent Q-Learning

```
Data: Initialize a global value function with random Q
Data: Initialize a single replay D_shared buffer for all agents
Data: Collect environment observations o_1^0, ..., o_n^0
for time step t = 0, ..., d do
    for agent i = 1, ..., n do
        if with probability ε then
            choose random action a_i
        else
            choose a_i = max_a Q(h_t, a_i)
        end
        Execute actions and collect observations o_i^{t+1} and rewards r_i
        Store all transitions in replay buffer D_shared
    end
    for agent i = 1, ..., n do
        Sample random mini-batch of transitions from replay buffer D_shared
        Update Q by the Bellman equation
    end
end
```

Algorithm 2.

Mean-field reinforcement learning

Mean-field reinforcement learning (MFRL) [Yan+18] offers a scalable and efficient approach to address the challenges in many-agent systems. In traditional MARL algorithms, the computational complexity can grow exponentially with the number of agents, making it impractical for systems with a large population. MFRL elegantly sidesteps this issue by approximating the many-body problem as a two-body problem. Here, each agent interacts not with all other agents, but with an average or "mean field" effect that represents the collective behaviour of the entire population. It can be resumed with this equation:

$$Q'(s, \boldsymbol{a}) = \frac{1}{N_j} \sum_{k \in N(j)} Q'(s, a', a^k)$$

Where N_j is the number of agents in the neighbourhood of agent j, and $\boldsymbol{a}$ is the joint action space of the agents in the neighbourhood of agent j. This approach is

particularly well-suited for CPSW with many behaviourally homogeneous agents. It aligns with the notion that complexity in such systems often arises from the interactions among simple agents, rather than from the complexity of individual agents. In fact, the mean-field effect can be seen as a form of communication between agents, where each agent is influenced by the collective behaviour of the entire population, but only through its local observations and *neighbourhood* interactions. In MFRL, the learning process is mutually reinforced between the individual agents and the dynamics of the overall population.

4.4 Final Remarks

This chapter provides a comprehensive overview of the foundational concepts in reinforcement learning. It begins by exploring the fundamental notion of intelligence and delves into key formalizations and algorithms that have shaped the field. This serves as a foundational understanding for the remainder of the paper, especially about the concept of CPSW. Indeed, we introduce swarMDP, a model that formalizes the dynamics of swarm-like systems, offering a theoretical framework for understanding complex agent interactions. Building on this, we present practical solutions to many-agent challenges through parameter sharing and experience sharing. Specifically, we introduce a unified approach known as many-agent Q-Learning. Lastly, we discuss the concept of mean-field reinforcement learning, emphasizing the critical role of neighbourhood context. Drawing parallels with aggregate computing, we argue that computation in these systems is inherently an interaction between an agent and its neighbours. This interaction fosters collective behaviours, leading to the overall system dynamics.

References

[ACS23] SV Albrecht, F Christianos, and L Schäfer. "Multi-Agent Reinforcement Learning: Foundations and Modern Approaches". In: *Massachusetts Institute of Technology: Cambridge, MA, USA* (2023).

[ACV22a] Gianluca Aguzzi, Roberto Casadei, and Mirko Viroli. "Addressing collective computations efficiency: Towards a platform-level reinforcement learning approach". In: *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE. 2022, pp. 11–20.

[ACV22b] Gianluca Aguzzi, Roberto Casadei, and Mirko Viroli. "Towards Reinforcement Learning-based Aggregate Computing". In: *Proc. of the Int. Conf. on Coordination Models and Languages*. Springer, 2022, pp. 72–91. DOI: 10.1007/978-3-031-08143-9_5.

[ACV23] Gianluca Aguzzi, Roberto Casadei, and Mirko Viroli. "MacroSwarm: A Field-Based Compositional Framework for Swarm Programming". In: *International Conference on Coordination Languages and Models*. Springer Nature Switzerland Cham. 2023, pp. 31–51.

[Agu+21] Gianluca Aguzzi et al. "ScaFi-Web: A Web-Based Application for Field-Based Coordination Programming". In: *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021*. Ed. by Ferruccio Damiani and Ornela Dardha. Vol. 12717. Lecture Notes in Computer Science. Springer, 2021, pp. 285–299. DOI: 10.1007/978-3-030-78142-2_18.

[Agu+22] Gianluca Aguzzi et al. "Dynamic Decentralization Domains for the Internet of Things". In: *IEEE Internet Computing* 26.6 (Nov. 2022), pp. 16–23. DOI: 10.1109/mic.2022.3216753.

[Aru+17] Kai Arulkumaran et al. "Deep Reinforcement Learning: A Brief Survey". In: *IEEE Signal Process. Mag.* 34.6 (2017), pp. 26–38. DOI: 10.1109/MSP.2017.2743240.

[Aud+16] Giorgio Audrito et al. “Run-Time Management of Computation Domains in Field Calculus”. In: *Foundations and Applications of Self* Systems, IEEE International Workshop on.* IEEE. 2016, pp. 192–197.

[Aud+18] Giorgio Audrito et al. “Space-time universality of field calculus”. In: *Coordination Models and Languages: 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings 20.* Springer. 2018, pp. 1–20.

[Aud+19a] Giorgio Audrito et al. “A Higher-Order Calculus of Computational Fields”. In: *ACM Transactions on Computational Logic* 20.1 (2019), 5:1–5:55. ISSN: 1529-3785.

[Aud+19b] Giorgio Audrito et al. “A Higher-Order Calculus of Computational Fields”. In: *ACM Trans. Comput. Log.* 20.1 (2019), 5:1–5:55. DOI: 10.1145/3285956.

[Aud+20] Giorgio Audrito et al. *Computation Against a Neighbour: Addressing Large-Scale Distribution and Adaptivity with Functional Programming and Scala.* 2020. DOI: 10.48550/ARXIV.2012.08626. URL: https://arxiv.org/abs/2012.08626.

[Aud+21] Giorgio Audrito et al. “Adaptive distributed monitors of spatial properties for cyber-physical systems”. In: *J. Syst. Softw.* 175 (2021), p. 110908. DOI: 10.1016/j.jss.2021.110908. URL: https://doi.org/10.1016/j.jss.2021.110908.

[Aud+22] Giorgio Audrito et al. “Functional Programming for Distributed Systems with XC”. In: *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany.* Ed. by Karim Ali and Jan Vitek. Vol. 222. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 20:1–20:28. DOI: 10.4230/LIPIcs.ECOOP.2022.20. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2022.20.

[Bea+08] Jacob Beal et al. “Fast self-healing gradients”. In: *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC).* ACM, 2008, pp. 1969–1975. DOI: 10.1145/1363686.1364163.

[Bea+12] Jacob Beal et al. “Organizing the Aggregate: Languages for Spatial Computing”. In: *CoRR* abs/1202.5509 (2012).

[Bea+17] Jacob Beal et al. “Self-adaptation to device distribution in the Internet of Things”. In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 12.3 (2017), pp. 1–29.

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

[Bra+13] Manuele Brambilla et al. "Swarm robotics: a review from the swarm engineering perspective". In: *Swarm Intell.* 7.1 (2013), pp. 1–41. DOI: 10.1007/s11721-012-0075-2. URL: https://doi.org/10.1007/s11721-012-0075-2.

[Cas+19] Roberto Casadei et al. "Engineering Resilient Collaborative Edge-Enabled IoT". In: *2019 IEEE International Conference on Services Computing, SCC 2019, Milan, Italy, July 8-13, 2019*. Ed. by Elisa Bertino et al. IEEE, 2019, pp. 36–45. DOI: 10.1109/SCC.2019.00019. URL: https://doi.org/10.1109/SCC.2019.00019.

[Cas+20a] Roberto Casadei et al. "FScaFi : A Core Calculus for Collective Adaptive Systems Programming". In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12477. Lecture Notes in Computer Science. Springer, 2020, pp. 344–360. DOI: 10.1007/978-3-030-61470-6_21. URL: https://doi.org/10.1007/978-3-030-61470-6%5C_21.

[Cas+20b] Roberto Casadei et al. "Pulverization in Cyber-Physical Systems: Engineering the Self-Organizing Logic Separated from Deployment". In: *Future Internet* 12.11 (2020), p. 203. DOI: 10.3390/fi12110203. URL: https://doi.org/10.3390/fi12110203.

[Cas+21a] Roberto Casadei et al. "Engineering collective intelligence at the edge with aggregate processes". In: *Eng. Appl. Artif. Intell.* 97 (2021), p. 104081. DOI: 10.1016/j.engappai.2020.104081. URL: https://doi.org/10.1016/j.engappai.2020.104081.

[Cas+21b] Roberto Casadei et al. "Tuple-Based Coordination in Large-Scale Situated Systems". In: *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Proceedings*. Vol. 12717. Lecture Notes in Computer Science. Springer, 2021, pp. 149–167. DOI: 10.1007/978-3-030-78142-2_10.

[Cas+22] Roberto Casadei et al. "Digital Twins, Virtual Devices, and Augmentations for Self-Organising Cyber-Physical Collectives". In: *Applied Sciences* 12.1 (2022). ISSN: 2076-3417. DOI: 10.3390/app12010349. URL: https://www.mdpi.com/2076-3417/12/1/349.

[Cas23] Roberto Casadei. “Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling”. In: *ACM Computing Surveys* (2023).

[CAV18] Roberto Casadei, Alessandro Aldini, and Mirko Viroli. “Towards attack-resistant Aggregate Computing using trust mechanisms”. In: *Sci. Comput. Program.* 167 (2018), pp. 114–137. DOI: `10.1016/j.scico.2018.07.006`. URL: `https://doi.org/10.1016/j.scico.2018.07.006`.

[CAV21a] Roberto Casadei, Gianluca Aguzzi, and Mirko Viroli. “A Programming Approach to Collective Autonomy”. In: *J. Sens. Actuator Networks* 10.2 (2021), p. 27. DOI: `10.3390/jsan10020027`.

[CAV21b] Roberto Casadei, Gianluca Aguzzi, and Mirko Viroli. “A programming approach to collective autonomy”. In: *Journal of Sensor and Actuator Networks* 10.2 (2021), p. 27.

[CV18] Roberto Casadei and Mirko Viroli. “Programming Actor-Based Collective Adaptive Systems”. In: *Programming with Actors - State-of-the-Art and Research Perspectives*. Ed. by Alessandro Ricci and Philipp Haller. Vol. 10789. Lecture Notes in Computer Science. Springer, 2018, pp. 94–122. DOI: `10.1007/978-3-030-00302-9_4`. URL: `https://doi.org/10.1007/978-3-030-00302-9%5C_4`.

[CV19] Roberto Casadei and Mirko Viroli. “Coordinating Computation at the Edge: a Decentralized, Self-Organizing, Spatial Approach”. In: *Fourth International Conference on Fog and Mobile Edge Computing, FMEC 2019, Rome, Italy, June 10-13, 2019*. IEEE, 2019, pp. 60–67. DOI: `10.1109/FMEC.2019.8795355`. URL: `https://doi.org/10.1109/FMEC.2019.8795355`.

[DBT00] Marco Dorigo, Eric Bonabeau, and Guy Theraulaz. “Ant algorithms and stigmergy”. In: *Future Gener. Comput. Syst.* 16.8 (2000), pp. 851–871. DOI: `10.1016/S0167-739X(00)00042-X`. URL: `https://doi.org/10.1016/S0167-739X(00)00042-X`.

[DH04] 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.

[DMC96] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. “Ant system: optimization by a colony of cooperating agents”. In: *IEEE Trans. Syst. Man Cybern. Part B* 26.1 (1996), pp. 29–41. DOI: `10.1109/3477.484436`. URL: `https://doi.org/10.1109/3477.484436`.

[Doe18] Sébastien Doeraene. "Cross-platform language design in Scala.js (keynote)". In: *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala, SCALA@ICFP 2018, St. Louis, MO, USA, September 28, 2018*. Ed. by Sebastian Erdweg and Bruno C. d. S. Oliveira. ACM, 2018, p. 1. DOI: 10.1145/3241653.3266230. URL: https://doi.org/10.1145/3241653.3266230.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000. ISBN: 0-262-04178-2.

[DV15] Ferruccio Damiani and Mirko Viroli. "Type-based self-stabilisation for computational fields". In: *Logical Methods in Computer Science* 11 (2015).

[FB16] Gianpiero Francesca and Mauro Birattari. "Automatic Design of Robot Swarms: Achievements and Challenges". In: *Frontiers Robotics AI* 3 (2016), p. 29. DOI: 10.3389/frobt.2016.00029. URL: https://doi.org/10.3389/frobt.2016.00029.

[Foe+17] Jakob N. Foerster et al. "Counterfactual Multi-Agent Policy Gradients". In: *CoRR* abs/1705.08926 (2017). arXiv: 1705.08926. URL: http://arxiv.org/abs/1705.08926.

[FS11] Francesco Riganti Fulginei and Alessandro Salvini. "The Flock of Starlings Optimization: Influence of Topological Rules on the Collective Behavior of Swarm Intelligence". In: *Computational Methods for the Innovative Design of Electrical Devices*. Ed. by Slawomir Wiak and Ewa Napieralska-Juszczak. Vol. 327. Studies in Computational Intelligence. 2011, pp. 129–145. DOI: 10.1007/978-3-642-16225-1_7. URL: https://doi.org/10.1007/978-3-642-16225-1%5C_7.

[Gel85] David Gelernter. "Generative Communication in Linda". In: *ACM Trans. Program. Lang. Syst.* 7.1 (1985), pp. 80–112. DOI: 10.1145/2363.2433. URL: https://doi.org/10.1145/2363.2433.

[Gu+21] Haotian Gu et al. "Mean-Field Multi-Agent Reinforcement Learning: A Decentralized Network Approach". In: *CoRR* abs/2108.02731 (2021). arXiv: 2108.02731. URL: https://arxiv.org/abs/2108.02731.

[HE10] Bernhard G. Humm and Ralf S. Engelschall. "Language-Oriented Programming Via DSL Stacking". In: *ICSOFT 2010 - Proceedings of the Fifth International Conference on Software and Data Technologies, Volume 2, Athens, Greece, July 22-24, 2010*. Ed. by José A. Moinhos Cordeiro, Maria Virvou, and Boris Shishkov. SciTePress, 2010, pp. 279–287.

[How60] Ronald A Howard. "Dynamic programming and markov processes." In: (1960).

[Hun13] John Hunt. "Cake Pattern". In: *Scala Design Patterns: Patterns for Practical Reuse and Design*. Cham: Springer International Publishing, 2013, pp. 115–119. ISBN: 978-3-319-02192-8. DOI: 10.1007/978-3-319-02192-8_13. URL: https://doi.org/10.1007/978-3-319-02192-8_13.

[KBP13] Jens Kober, J. Andrew Bagnell, and Jan Peters. "Reinforcement learning in robotics: A survey". In: *Int. J. Robotics Res.* 32.11 (2013), pp. 1238–1274. DOI: 10.1177/0278364913495721.

[KE95] James Kennedy and Russell Eberhart. "Particle swarm optimization". In: *Proceedings of International Conference on Neural Networks (ICNN'95), Perth, WA, Australia, November 27 - December 1, 1995*. IEEE, 1995, pp. 1942–1948. DOI: 10.1109/ICNN.1995.488968. URL: https://doi.org/10.1109/ICNN.1995.488968.

[KMP13] Soummya Kar, José M. F. Moura, and H. Vincent Poor. "QD-Learning: A Collaborative Distributed Strategy for Multi-Agent Reinforcement Learning Through *Consensus + Innovations*". In: *IEEE Trans. Signal Process.* 61.7 (2013), pp. 1848–1862. DOI: 10.1109/TSP.2013.2241057. URL: https://doi.org/10.1109/TSP.2013.2241057.

[LLM15] Alberto Lluch Lafuente, Michele Loreti, and Ugo Montanari. "A fixpoint-based calculus for graph-shaped computational fields". In: *Coordination Models and Languages: 17th IFIP WG 6.1 International Conference, COORDINATION 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings 17*. Springer. 2015, pp. 101–116.

[Low+17] Ryan Lowe et al. "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al. 2017, pp. 6379–6390. URL: https://proceedings.neurips.cc/paper/2017/hash/68a9750337a418a86fe06c1991a1d64c-Abstract.html.

[Luo+19] Nguyen Cong Luong et al. "Applications of Deep Reinforcement Learning in Communications and Networking: A Survey". In: *IEEE Commun. Surv. Tutorials* 21.4 (2019), pp. 3133–3174. DOI: 10.1109/COMST.2019.2916583.

[LW22] Xinyang Li and Yifan Wang. "Zero-Sum Stochastic Stackelberg Games". In: *arXiv preprint arXiv:2211.13847* (2022).

[MLF07] Laëtitia Matignon, Guillaume J. Laurent, and Nadine Le Fort-Piat. "Hysteretic q-learning : an algorithm for decentralized reinforcement learning in cooperative multi-agent teams". In: *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems, October 29 - November 2, 2007, Sheraton Hotel and Marina, San Diego, California, USA*. IEEE, 2007, pp. 64–69. DOI: 10.1109/IROS.2007.4399095. URL: https://doi.org/10.1109/IROS.2007.4399095.

[Mni+13] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[MZL04] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. "Co-Fields: A Physically Inspired Approach to Motion Coordination". In: *IEEE Pervasive Computing* 3.2 (2004), pp. 52–61. DOI: 10.1109/MPRV.2004.1316820.

[NKC+09] Ngoc Thanh Nguyen, Ryszard Kowalczyk, Shyi-Ming Chen, et al. *Computational Collective Intelligence. Semantic Web, Social Networks and Multiagent Systems: First International Conference, ICCCI 2009, Wrocław, Poland, October 5-7, 2009. Proceedings.* Springer, 2009.

[NS03] Abraham Neyman and Sylvain Sorin. *Stochastic games and applications*. Vol. 570. Springer Science & Business Media, 2003.

[NW04] Ryan Newton and Matt Welsh. "Region streams: Functional macroprogramming for sensor networks". In: *Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004*. 2004, pp. 78–87.

[OH19] Afshin Oroojlooyjadid and Davood Hajinezhad. "A Review of Cooperative Multi-Agent Deep Reinforcement Learning". In: *CoRR* abs/1908.03963 (2019). arXiv: 1908.03963. URL: http://arxiv.org/abs/1908.03963.

[OZ05] Martin Odersky and Matthias Zenger. "Scalable component abstractions". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. Ed. by Ralph E. Johnson and Richard P. Gabriel. ACM, 2005, pp. 41–57. DOI: 10.1145/1094811.1094815. URL: https://doi.org/10.1145/1094811.1094815.

[Pia+21] Danilo Pianini et al. "Time-Fluid Field-Based Coordination through Programmable Distributed Schedulers". In: *Log. Methods Comput. Sci.* 17.4 (2021). URL: https://doi.org/10.46298/lmcs-17(4:13)2021.

[PMV13] D Pianini, S Montagna, and M Viroli. "Chemical-oriented simulation of computational systems with ALCHEMIST". In: *J. of Simulation* 7.3 (2013), pp. 202–215. DOI: 10.1057/jos.2012.27.

[Ras+18] Tabish Rashid et al. "QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 4292–4301. URL: http://proceedings.mlr.press/v80/rashid18a.html.

[RBW15] Raymond Roestenburg, Rob Bakker, and Rob Williams. *Akka in Action.* 1st. USA: Manning Publications Co., 2015. ISBN: 1617291013.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[Sch+17] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[Šoš+16] Adrian Šošic et al. "Inverse reinforcement learning in swarm systems". In: *arXiv preprint arXiv:1602.05450* (2016).

[Sut+99] Richard S. Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*. Ed. by Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller. The MIT Press, 1999, pp. 1057–1063. URL: http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.

[Sut88] Richard S. Sutton. "Learning to Predict by the Methods of Temporal Differences". In: *Mach. Learn.* 3 (1988), pp. 9–44. DOI: 10.1007/BF00115009. URL: https://doi.org/10.1007/BF00115009.

[Tan93a] Ming Tan. "Multi-Agent Reinforcement Learning: Independent versus Cooperative Agents". In: *Machine Learning, Proceedings of the Tenth International Conference, University of Massachusetts, Amherst, MA, USA, June 27-29, 1993*. Ed. by Paul E. Utgoff. Morgan Kaufmann, 1993, pp. 330–337. DOI: `10.1016/b978-1-55860-307-3.50049-6`. URL: `https://doi.org/10.1016/b978-1-55860-307-3.50049-6`.

[Tan93b] Ming Tan. "Multi-agent reinforcement learning: Independent vs. cooperative agents". In: *Proceedings of the tenth international conference on machine learning*. 1993, pp. 330–337.

[Tes+22] Lorenzo Testa et al. "Aggregate processes as distributed adaptive services for the Industrial Internet of Things". In: *Pervasive and Mobile Computing* 85 (2022), p. 101658. ISSN: 1574-1192. DOI: `https://doi.org/10.1016/j.pmcj.2022.101658`. URL: `https://www.sciencedirect.com/science/article/pii/S1574119222000797`.

[Tur50] Alan M. Turing. "Computing Machinery and Intelligence". In: *Mind* 59.October (1950), pp. 433–60. DOI: `10.1093/mind/lix.236.433`.

[VDB13] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. "A calculus of computational fields". In: *Advances in Service-Oriented and Cloud Computing: Workshops of ESOCC 2013, Málaga, Spain, September 11-13, 2013, Revised Selected Papers 2*. Springer. 2013, pp. 114–128.

[Vir+18a] Mirko Viroli et al. "Engineering Resilient Collective Adaptive Systems by Self-Stabilisation". In: *ACM Trans. Model. Comput. Simul.* 28.2 (2018), 16:1–16:28. DOI: `10.1145/3177774`. URL: `https://doi.org/10.1145/3177774`.

[Vir+18b] Mirko Viroli et al. "Engineering resilient collective adaptive systems by self-stabilisation". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 28.2 (2018), pp. 1–28.

[Vir+19] Mirko Viroli et al. "From distributed coordination to field calculus and aggregate computing". In: *J. Log. Algebraic Methods Program.* 109 (2019). DOI: `10.1016/j.jlamp.2019.100486`. URL: `https://doi.org/10.1016/j.jlamp.2019.100486`.

[War89] Charles W. Warren. "Global path planning using artificial potential fields". In: *IEEE Conf. on Robotics and Automation*. 1989. DOI: `10.1109/ROBOT.1989.100007`.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. "Technical Note Q-Learning". In: *Mach. Learn.* 8 (1992), pp. 279–292. DOI: `10.1007/BF00992698`.

[WH07] Tom De Wolf and Tom Holvoet. "Designing Self-Organising Emergent Systems based on Information Flows and Feedback-loops". In: *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, Boston, MA, USA, July 9-11, 2007*. IEEE Computer Society, 2007, pp. 295–298. DOI: 10.1109/SASO.2007.16. URL: https://doi.org/10.1109/SASO.2007.16.

[Wik23a] Wikipedia contributors. *2017 Turin stampede — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-October-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=2017_Turin_stampede&oldid=1164182872.

[Wik23b] Wikipedia contributors. *2023 Canadian wildfires — Wikipedia, The Free Encyclopedia*. [Online; accessed 5-October-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=2023_Canadian_wildfires&oldid=1178342069.

[Yan+18] Yaodong Yang et al. *Mean Field Multi-Agent Reinforcement Learning*. 2018.

[Yan21] Yaodong Yang. "Many-agent reinforcement learning". PhD thesis. UCL (University College London), 2021.

[Yu+21] Chao Yu et al. "The Surprising Effectiveness of MAPPO in Cooperative, Multi-Agent Games". In: *CoRR* abs/2103.01955 (2021). arXiv: 2103.01955. URL: https://arxiv.org/abs/2103.01955.

Part II

Engineering Cyber-Physical Swarms

Chapter 5

Patterns: Sensing-driven Clustering in Swarms

What is sensing-driven clustering?
How this problem can be addressed in CPSW?
What are the *assumptions* underlying the approach?
Why is this problem relevant for CPSW?
– **RQ3, RQ4**

Contents

5.1 Field-based Concurrent Processes 90
5.2 Resilient Dynamic Cluster Formation 92
5.3 Sensing-Driven Clustering 94
5.3.1 Assumptions 94
5.3.2 Problem Definition 95
5.3.3 Adaptive Centroid-based Clustering on Numeric Values 97
5.3.4 Adaptive Clustering Meta-Algorithm 98
5.4 Evaluation 102
5.4.1 Scenario Description 102
5.4.2 Evaluation Goals 103
5.4.3 Simulation Framework 104
5.4.4 Simulations 106
5.4.5 Results 111

5.4.6 Discussion 113
5.5 Related Work 114
5.5.1 Swarm-based Environment Monitoring 115
5.5.2 Related Clustering Models and Problems 115
5.5.3 Related Work on Sensing-based Clustering 117
5.5.4 Related Approaches and Programming Models 118
5.5.5 Related Field-based Algorithms 118
5.6 Final Remarks 119

Swarm intelligence is the collective-level ability to solve problems in large groups of relatively simple agents that interact with each other locally, i.e., based on physical/logical proximity [BDT99]. Common but not exhaustive classes of collective behaviours include spatial organization (e.g., pattern formation), swarm navigation, and collective-decision making [Bra+13].

In particular, one problem of interest is *swarm clustering* [LKK05; CNM17], whereby the classical data clustering task (i.e., the unsupervised learning task where data items are grouped to promote intra-group similarity) is brought in swarm settings. This problem revolves around splitting the swarm into groups of individuals, called *clusters*, such that the individuals in the same *cluster* are more similar to each other (for some definition of *similarity*) than to those in other clusters. Once a cluster is formed, typically it is assigned a *sub-goal* to be carried on collectively. Typical clustering approaches may consider the spatial distribution of the individuals or the goals of the individuals to define clusters that represent, e.g., teams or interaction domains. In this chapter, we focus on *sensing-based clustering* [LM07], namely a clustering problem that considers both the spatial distribution of individuals and the environmental values sensed by these individuals (through sensors). This is essential for CPSW applications due to the tight integration with the physical world. That is, the goal is to seek clusters of neighbour individuals with a similar perception of some sensed value. The problem can be in a *static* form, where a snapshot of the system state is considered, or in a *dynamic* form, where values change over time and solutions have to deal with change somehow. The problem has been considered in Wireless Sensor Networks (WSNs) and Internet-of-Things (IoT) applications like environment monitoring and control [LM07], efficient distributed collection [Pha+10], and disaster management [Kuc+20]. However, to the best of our knowledge, no existing work addresses the dynamic problem in CPSW, which requires specific techniques to adaptively re-adjust clusters to face changes. Additionally, we look for solutions featuring *resilience*, namely, leveraging distribution and decentralization to continuously face changes and faults, hence avoiding single points of failures and

potential bottlenecks. Accordingly, we present and address the *dynamic sensing-based swarm clustering* problem, based on our language-based view of CPSW, namely enforcing the use of a field-based programming model, both for the specification of the problem (i.e., the system model) and for the solution (i.e., the distributed aggregate computing program definition).

Essentially, the core idea of our clustering approach is to make agents in local minima (or maxima) of the sensed value (depending on whether lowest or highest values are most significant) spawn a spatial process of gathering for neighbour devices until finding the proper size of the cluster, additionally managing interactions with other clusters when there are overlaps.

5.1 Field-based Concurrent Processes

Field-based concurrent processes, also called *aggregate processes* [Cas+19; Cas+21], are field-based computations that exist dynamically: they can be dynamically generated (usually by individual agents), execute on a dynamic set of agents, and disappear once all its members withdraw. They have been formalized in [Cas+19] and deeply covered in [Cas+21], showing how they can support the design of intelligent collective behaviour by extending the practical expressiveness of field-based programming models [Vir+19]. We provide a brief account of the details relevant for this chapter in the following.

Indeed, the aggregate process abstraction is relevant since an aggregate process instance, by running on a (evolving) subset of the agents, can be used to denote a *dynamic* cluster. Therefore, clustering algorithms can be expressed in terms of how aggregate processes are generated (candidate cluster formation) and merged/removed (cluster selection).

Aggregate processes can be expressed as normal field-based functions and spawned through a `spawn` construct with the following signature:

```
// spawn is a generic function which accepts 3 parameters
def spawn[K,A,R](process: K => A => (R,Boolean),
                 newProcesses: Set[K],
                 args: A): Map[K,R]
```

The generic type `K` instantiates to the type of *process key*, also called a *process identifier (PID)*, which also works as construction parameter; the generic type `A` instantiates to the type of runtime parameters for the currently running process instances; the generic type `R` instantiates to the type of the output of the process. A *process definition* has curried type `K =>A =>(R,Boolean)`, namely a function from a value of type `K` and a value of type `A` to a pair of a value of type `R` and a Boolean. The Boolean value, called the *process status*, expresses if the device

that has executed a given process instance would like to participate in the process (`true` status) or not (`false` status). The crucial point is that every device that participates in a process with PID π automatically propagates the process PID π to all its neighbours, which will run a corresponding process instance when the `spawn` function is evaluated. So, the `spawn` function accepts a function `process` of a field-based behaviour, a set `newProcesses` of new process instances to be generated locally in the current round, and a value of type `A` for the runtime input of the instances currently running in the local round of a given device. Notice that, though `process` can be a field of functions, it is typically a constant field of the same function, which means that usually a `spawn` expression enables running zero or more process instances of the same kind of process. Evaluation of `spawn` returns a `Map[K,R]` (i.e., a hashmap or dictionary) which is a set of entries mapping the PIDs of executed process instances (with status `true`) to corresponding outputs of type `R`.

As an example, consider building a separate gradient computation for each distinct source agent, that will expand within a certain range ρ. This could be coded as follows in ScaFi:

```
type DeviceId = Int
// Process definition as a function
val proc: DeviceId => Boolean => (Double, Boolean) = id => isSource => {
  val output = gradient(id == deviceId())
  val status = if(id == deviceId()) isSource
               else output < ρ
  (output, status)
}
// Set of processes to be generated locally
val newProcesses: Set[DeviceId] =
  if(isSource()) Set(deviceId()) else Set.empty
// Expression for handling acquired and generated processes
val gradients: Map[DeviceId,Double] =
  spawn[DeviceId,Boolean,Double](process, newProcesses, isSource())
```

In detail, the IDs of sources are used as PIDs; so, for instance, a gradient from agent 7 will become a process with PID 7. The process logic is defined through `proc`, which is a function of the PID `id` and Boolean argument `isSource` denoting whether the running agent is a source, as provided by built-in sensor function `isSource()`. In `proc`, a gradient is built from the agent whose ID, provided by `deviceId()`, matches the `id` of the source corresponding to the current process instance. Then, `status` is defined `true` if the source for the process is still a source or, for non-source agents, if their gradient value is lower than threshold ρ. Notice that when the original source is not a source any more, the gradient

`output` will rise, eventually causing all the agents to leave that process. Value `newProcesses` will be a singleton set with the ID of the running device when its `isSource()` sensor returns true, or the empty set otherwise. In the former case, a corresponding process is spawned if it did not already exist. The evaluation of the `spawn` call, then, will run both new and existing processes including those executed (and not quit) at the previous round, as well as those acquired from neighbours. The output of the `spawn` expression will be a map from the PIDs of the processes locally executed to the value of the gradient (`output`) locally computed in those process instances.

An example of the dynamics of such a program is provided in Figure 5.1. In the picture: nodes are agents; labels on nodes are agent IDs; edges denote neighbouring links, over which messages are sent and received; the output of the `spawn` expression is shown above the nodes unless it is an empty map (not shown); the different sub-pictures are snapshots of a corresponding hypothetical system state trajectory that may result after multiple rounds of execution in multiple devices. A more thorough introduction and description of aggregate processes together with more examples is available in [Cas+21].

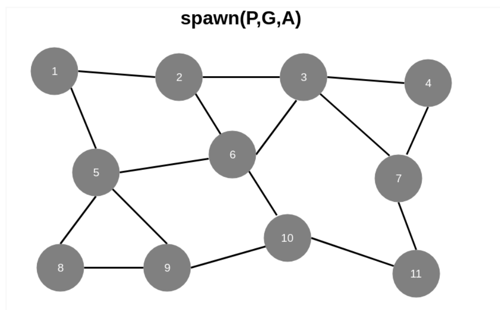
5.2 Resilient Dynamic Cluster Formation

Different cluster models exist and, for each cluster model, several algorithms can be devised [Est02]. These are reviewed and compared with our cluster model in Section 5.5.

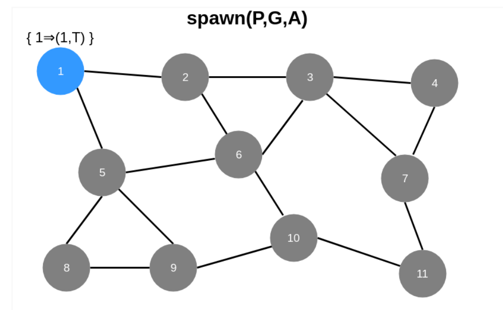
In this chapter, we focus on *swarm clustering*, which involves associating each swarm member to zero or more clusters. So, this is a problem of *cluster formation* [GHZ18], more than a problem of *cluster analysis* (which generally includes cluster formation followed by cluster evaluation). A *cluster*, in this setting, is essentially a label (*cluster ID*), which can be associated to an agent, and that can be used to determine its behaviour. In field terms, a clustering can be seen as a field mapping each agent to a set of cluster IDs—we call this a *clustering field*.

Essentially, a cluster can be used to determine, query, and control a group of agents. Such a group could represent a team, used for cooperation or to solve a common goal, or a space-time domain for a field computation. Indeed, as the agents are situated in space, they provide a means for extracting data from their corresponding location, which may be instrumental for environment monitoring, data acquisition, etc.

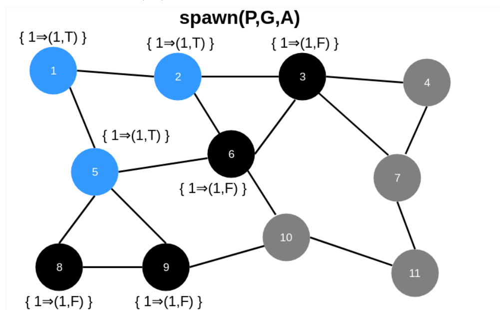
Moreover, we consider *dynamic clustering* [RTG19], where the emphasis is not on identifying a single clustering for a given system configuration, but to update and evolve a clustering solution as the system configuration evolves (e.g., due to mobility, failure, or change in other clustering criteria). The specific problem we



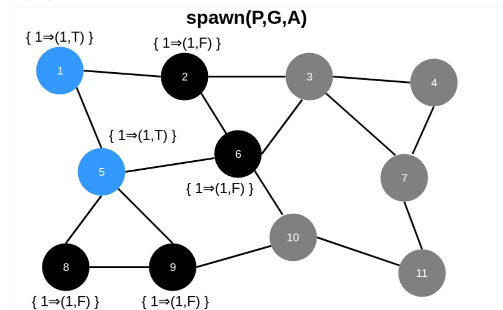
(a) Initial network.



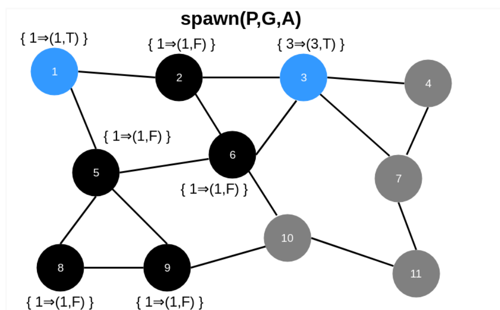
(b) A process is generated on agent 1.



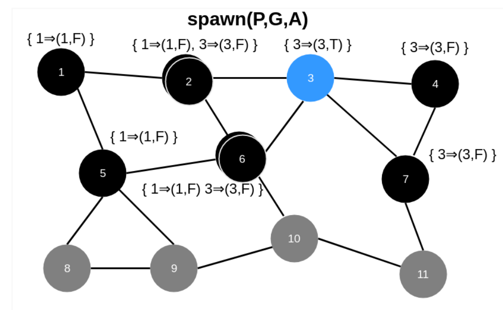
(c) The process with PID 1 propagates up to a certain range.



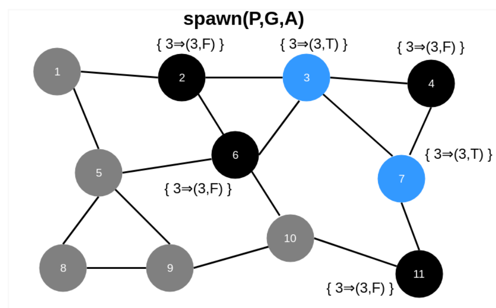
(d) The "border" of a process can change dynamically.



(e) Another process is spawned by source agent 3.



(f) Processes can overlap. Agents 2 and 6 run the two processes with PID 1 and 3.



(g) Process 1 ceases to exist.

Figure 5.1: Examples of the dynamics of multiple concurrent gradient processes.

tackle is *dynamic sensing-based/space-based swarm clustering*, which involves associating each swarm member to zero or more clusters, and to evolve such association by considering change in the environment (*sensing-based*) and spatial location of the members (*space-based*).

In summary, our goal is to define a distributed, decentralized, field-based clustering algorithm, for the system model based on our language-based view described in Section 3.3.1, able to create and dynamically maintain a clustering field, resiliently. Our focus on resilience make centralized approaches not appropriate since we cannot assume that some nodes are infallible or always available. This work draws motivation from (i) the relevance of the problem for *situated* systems as CPSW, (ii) a scarcity of solutions to the problem of *sensing-driven spatial clustering* in literature, and (iii) a general lack of effective field-based clustering solutions. Refer to Section 5.5 for a more detailed account on these research gaps.

5.3 Sensing-Driven Clustering

In this section, after describing a minimal set of *assumptions* underlying the approach (Section 5.3.1), we define the *problem* to be addressed (Section 5.3.2), in terms of inputs, outputs, and parameters, describe a specific instantiation of the problem for *centroid-based* clustering on numeric values (Section 5.3.3), and then present a meta-algorithm providing a solution to the stated problem (Section 5.3.4).

5.3.1 Assumptions

Before formally defining the problem of *Dynamic Sensing Based Swarm Clustering*, we summarize the assumptions about the swarm devices and the environment in which they act. Such assumptions justify both the way we define the problem, and some of the design choices we adopt for its solution.

1. A *swarm* is composed by a set of possibly many relatively simple autonomous cyber-physical agents,
2. An *agent* can move within the environment, sense, and actuate.
3. *Communication* is based on peer-to-peer connection link, based on the proximity of agents, without relying on infrastructure (e.g., LTE network, Wi-Fi network).
4. *Reliability* of agents themselves and communications are not guaranteed and, in some scenarios, failures are quite likely.

5. The measures of the *environment*, as sensed by the agents, can change over time.

6. The measures of interest of the *environment* at two points in space close to each other tend to be positively correlated.

The above assumptions, based on our system model defined in Part I, are rather weak and, therefore, quite challenging. They encompass scenarios where a swarm of agents explores an area where multiple natural phenomena are happening. Therefore, even if the proposed solution target CPSW, it can be applied to other scenarios, such as WSNs and IoT, where the above assumptions are satisfied.

The field-based clustering algorithm for solving the *Dynamic Sensing Based Swarm Clustering* problem will be discussed below. For now, we just want to point out that our assumptions justify a fully distributed approach in which agents exchange information with their neighbours.

First, given the very nature of swarm systems, problems are usually better solved by distributed algorithms than centralized algorithms, e.g., [Hos13; CNM17]. In particular, by our assumption that agents and communications can fail, and that there is no global communication infrastructure, a node in charge of all the computations (either an agent or a base station) would constitute a risky single point of failure. Even if the swarm was able to recover from such failure by automatically choosing another central node, the switch would be cumbersome and potentially very costly, only to reach again a situation with another single point of failure.

Given agents whose connection links are established and lost based on the proximity with other agents, it may be possible to build an abstraction on top of that, whereby multi-hop communications are transparent, and each agent has the illusion of being able to immediately communicate with any other agent in the swarm by specifying an appropriate ID (this is, e.g., the typical abstraction brought by the IP layer of the TCP/IP stack). While the cost of adding such an additional layer may be acceptable in some situations, for the specific goal of clustering this would not bring any advantage: as we shall see in the sections below, clusters spring out, expand, and collapse following spatial vicinity—i.e., a new cluster expands first to the immediate neighbours of the agent that generated it, and then progressively incrementally spreads to further agents.

5.3.2 Problem Definition

We address the problem of situation awareness and recognition, where a value distributed in space (e.g., temperature as measured by sensors) has to be monitored, by recognizing compact clusters with similar values (e.g., spatial regions with a

similar temperature). This problem, called *sensing-driven clustering* in literature, has been investigated largely in static scenarios [Kuc+20; Pha+10; LM07], where data from a fixed sensor network has to be processed to obtain the relevant clusters. However, solutions for such networks do not extend well to dynamic contexts, such as a mobile CPSWs system monitoring an environment: in this scenario, mobility and proximity of communication are key, and need to be handled by an algorithm that is resilient to changes in values, network structure and placement in space. To the best of our knowledge, this problem has never been previously considered in the literature.

A sensing-driven clustering algorithm for such system could be useful for several outcomes. Clusters may provide a compressed summary of the value distribution in space, to upload on the cloud and be graphically represented for human convenience. Clusters may also be used to drive more complex situation recognition patterns: algorithms to detect dangerous situations may be run in each cluster separately, using information from that cluster to reach a verdict, without interference from information on neighbouring clusters. Clusters may also be used to drive task assignments to the monitoring drones, possibly guiding their placement in space, by directing more drones in clusters where the need arises.

More formally, we consider the following problem:

- **Input:** for each device, a unique identifier i and a *value* v_i of type T (possibly obtained through a sensor reading);
- **Output:** for each device, a list of clusters to which the device belongs, represented as a map from unique identifiers l of cluster leaders to corresponding cluster summary values w^l of type S.

To formally specify the output, we need some further details characterizing what a *cluster* is, how they should be selected, and what is their *summary*. This is attained through the following problem parameters.

- **Metric:** a data type M with
  - a *null* value 0_M;
  - a partial order[1] $x \leq y$ defined for x, y of type M;
  - an addition operator $x+y$ defined for x, y of type M, such that $x+0_M = x$ and $x + y > x$ if $y > 0_M$;
  - a positive function $d(i, j) > 0_M$ returning a value in M representing a distance between a device i and j (depending on the devices' sensor

[1]A partial order is a reflexive, transitive and anti-symmetric relation; with no requirement that either $x \leq y$ or $y \leq x$ for x, y of type M.

states and possibly values v_i). This is intended to make use of spatial distance estimates as well as other factors (i.e., value distances).

- **Summary:** a data type S with
  - a value $s(i)$ of type S in every device i (depending on sensor state);
  - an associative and commutative function $f : (S, S) \to S$, used to aggregate values $s(i)$ for devices in a same cluster.
- **Leader selection:**
  - a candidate radius $r(i)$ in M (depending on sensor state and values), so that only devices with a relative distance strictly lower than $r(i)$ can belong to a cluster whose leader is i;[2]
  - a commutative similarity predicate $p : (S, S) \to \{\top, \bot\}$, identifying similar clusters based on their summary.

According to this description, a candidate cluster $\mathcal{C}$ is a set of devices with a leader i, such that every $j \in \mathcal{C}$ is within a distance of $r(i)$ from the leader i, according to the metric given by d. The summary w_i of such cluster is the repeated aggregation through f of the values $\{v_j : j \in \mathcal{C}\}$. Nearby clusters are merged if their summaries are similar according to predicate p, and in such cases, the lowest identifier is selected as the leader of the merged cluster.

Leaders are used to regulating clusters via aggregate processes and to easily support consistent coordination and decision-making regarding the activity of a cluster. Notice that agents may belong to multiple clusters: this is important to support tracking phenomena that are spatially close to each other. Indeed, if a node is in between two phenomena, it could participate in the corresponding clusters at the same time to help to track or handle both phenomena. We highlight that we aim to solve this problem by an *adaptive* algorithm, that is, a program that is able to handle changes in its input, by periodically and asynchronously updating its internal values.

5.3.3 Adaptive Centroid-based Clustering on Numeric Values

In the evaluation section, we consider a specific instantiation of the parameters just introduced, for centroid-based clustering on numeric values. In this context, the metric is a simple distance on values, so that $d(i, j) = |v_i - v_j|$. To prevent the creation of a candidate cluster for every device, the candidate radius $r(i)$ is set

[2]Notice that $r(i) = 0_M$ implies that no device can be in a cluster whose leader is i.

to zero whenever i is not a local minimum (i.e., has a neighbouring device j such that $v_j < v_i$). If instead i is a local minimum, $r(i)$ is set to a fixed difference value θ. The values $s(i)$ to be summarized are set to a tuple $[x_i, y_i, v_i, 1]$ of the devices' positions[3] and values with the number 1, with an aggregator function f that is a component-wise sum, so that the overall aggregate of a cluster $\mathcal{C}$ is (eventually) equal to the tuple $[\sum_{i\in\mathcal{C}} x_i, \sum_{i\in\mathcal{C}} y_i, \sum_{i\in\mathcal{C}} v_i, \#\mathcal{C}]$ (where $\#\mathcal{C}$ is the actual number of members of cluster C). The similarity predicate p then declares two clusters as similar if they have centroids within a radius of γ, in a 3D space mixing spatial coordinates with a value coordinate:

$$p([x, y, v, n], [x', y', v', n']) \quad := \quad \|\frac{(x, y, v)}{n} - \frac{(x', y', v')}{n'}\| < \gamma$$

where (x, y, v) denotes a 3D vector and $\|\cdot\|$ denotes the norm of a vector. By setting the problem parameters as described, the meta-algorithm can select clusters of similar value, led by their minima, and merge overlapping clusters that are too close together and with a similar value.

5.3.4 Adaptive Clustering Meta-Algorithm

We now describe the general meta-algorithm for the stated problem through state equations. The algorithm state is distributed, hence composed of variables x_i depending on a device identifier i: we assume that such a variable is stored in device i and periodically updated by it through the state equations. Each equation may involve inspecting the state of variables in neighbour devices j: we assume that every device periodically shares its state with neighbours, so that a (not necessarily updated) view of neighbours' state is available in each device, and each state equation can be computed locally in the current device i, without remote memory accesses. We use $\mathcal{N}(i)$ to denote the set of current neighbours of device i, i.e., the set of devices j for which a view of their state is locally available in i (not including i itself). The execution of state equations can be performed in asynchronous rounds, as described in Part I. In order to showcase the algorithm at work by examples, in the following we consider a network of three interconnected devices $i = 0, 1, 2$, so that $\mathcal{N}(0) = \mathcal{N}(1) = \mathcal{N}(2) = \{0, 1, 2\}$. We assume that the devices are placed in positions $(x_0, y_0) = (0, 0)$, $(x_1, y_1) = (1, 1)$, $(x_2, y_2) = (2, 0)$ and hold values $v_0 = 2$, $v_1 = 3$, $v_2 = 1$. We will also assume that the parameters are as described in Section 5.3.3, with $\theta = \gamma = 3$.

Table 5.1 summarizes the state variables used in state equations. Every device maintains a candidate leader set $\mathcal{S}_i$, of possible clusters to which the device may

[3]We assume that a GPS-like sensor is available.

i	current device	$\mathcal{N}(i)$	neighbour set
ℓ	candidate leader	$\mathcal{S}_i$	candidate leader set
m_i^ℓ	metric in i from ℓ	c_i^ℓ	whether i belongs to cluster ℓ
p_i^ℓ	parent of i in cluster ℓ	t_i^ℓ	partial summary in i for cluster ℓ
u_i^ℓ	candidate leader summary in i for ℓ		
l_i	selected leader for cluster i, if any	w_i	selected summary for cluster i, if any
l_i^ℓ	selected leader for cluster ℓ in i	w_i^ℓ	selected summary for cluster ℓ in i

Table 5.1: State variables used in the state equations.

belong. Every round, this set is updated as:

$$\mathcal{S}_i \;=\; \{\ell \in \mathcal{S}_j \text{ for } j \in \mathcal{N}(i) \text{ s.t. } c_j^\ell = \top\} \cup \begin{cases} \emptyset & \text{if } r(i) = 0_M \\ \{i\} & \text{otherwise} \end{cases}$$

Thus, $\mathcal{S}_i$ includes i provided that $r(i) > 0_M$, together with other candidate leaders ℓ considered by neighbours (in their candidate leader set and which have computed to be within the cluster). In field-based computing, this set is implicitly maintained by the *spawn* construct, given c_i^ℓ as process return status and $\{i\}$ as new process key (if $r(i) > 0_M$). In our sample network, the initial value for $\mathcal{S}_i$ in each i will only consider the current device, as information from neighbouring devices is not available yet. Thus, we will have $\mathcal{S}_0 = \{0\}, \mathcal{S}_1 = \{\}, \mathcal{S}_2 = \{2\}$. After convergence, each node will understand itself as possibly belonging to clusters 0 and 2, so that $\mathcal{S}_0 = \mathcal{S}_1 = \mathcal{S}_2 = \{0, 2\}$.

Most of the meta-algorithm computation is repeated for each of the candidate leaders $\ell \in \mathcal{S}_i$. First, a metric m_i^ℓ of the distance between ℓ and i is computed, through the following equation (i.e., the `gradient` block in field-based computing—cf. Section 3.3.2.3):

$$m_i^\ell = \begin{cases} 0_M & \text{if } \ell = i \\ \min\{m_j^\ell + d(i,j) : \; j \in \mathcal{N}(i)\} & \text{otherwise} \end{cases}$$

In the sample network, we will have $m_0^0 = m_2^2 = 0$, $m_1^0 = 0 + |v_0 - v_1| = 1$, $m_1^2 = 0 + |v_2 - v_1| = 2$, $m_0^2 = m_2^0 = 0 + |v_0 - v_1| + |v_2 - v_1| = 3$. From m_i^ℓ, we also decide the values c_i^ℓ as the truth predicates of whether $m_i^\ell \leq \theta$.

Then, an optional parent p_i^ℓ for $\ell \neq i$ is determined as the neighbour j with minimal m_j^ℓ (resolving ties by the identifier j itself):

$$p_i^\ell = \begin{cases} \arg\min_{j \in \mathcal{N}(i)} \{(m_j^\ell, j)\} & \text{if } \ell \neq i \\ \text{None} & \text{otherwise} \end{cases}$$

In our example, we have that $p_1^0 = 0$, $p_1^2 = 2$, $p_0^2 = p_2^0 = 1$ while p_0^0 and p_2^2 are undefined. Through it, partial summaries t_i^ℓ can be computed (`C` block in field-based computing—cf. Section 3.3.2.3):

$$t_i^\ell = \text{reduce}(\{s(i)\} \cup \{t_j^\ell : \ j \in \mathcal{N}(i) \text{ and } p_j^\ell = i\}, f)$$

where "reduce" is a function accumulating every element of a given set with the given binary function, and thus aggregates with f the value $s(i)$ together with the t_j^ℓ values of neighbours j which chose the current device i as their parent. In the sample network, we will have that $t_0^2 = s(0) = (0, 0, 2, 1)$, $t_2^0 = s(2) = (2, 0, 1, 1)$, $t_1^0 = s(1) + s(2)$, $t_1^2 = s(1) + s(0)$, $t_0^0 = t_2^2 = s(0) + s(1) + s(2) = (3, 1, 6, 3)$. The value of the partial summary in the leader is then propagated through the cluster by a broadcast function:

$$u_i^\ell = \begin{cases} t_i^\ell & \text{if } \ell = i \\ u_{p_i^\ell}^\ell & \text{otherwise} \end{cases}$$

so that, in our example after convergence, each u_i^ℓ is $(3, 1, 6, 3)$. Every candidate leader i with $r(i) > 0_M$ is now able to choose its selected leader l_i, as the minimum candidate leader j (possibly i itself) with a summary similar to that of i according to predicate p:

$$(l_i, w_i) = \begin{cases} \min\{(\ell, u_i^\ell) : \ \ell \in \mathcal{S}_i \text{ and } p(u_i^\ell, u_i^i)\} & \text{if } r(i) > 0_M \\ \text{None} & \text{otherwise} \end{cases}$$

In the running example, we will have that $l_0 = l_2 = 0$, $w_0 = w_2 = (3, 1, 6, 3)$, since the two clusters are fully overlapping hence p is true. The selected leader l_i and corresponding summary w_i is then propagated by broadcast through the cluster of i. For every $\ell \in \mathcal{S}_i$:

$$(l_i^\ell, w_i^\ell) = \begin{cases} (l_i, w_i) & \text{if } \ell = i \\ (l_{p_i^\ell}^\ell, w_{p_i^\ell}^\ell) & \text{otherwise} \end{cases}$$

Finally, in every device i, the meta-algorithm output is the map:

$$\{l_i^\ell \mapsto w_i^\ell : \ \ell \in \mathcal{S}_i\}.$$

This meta-algorithm is presented as ScaFi pseudo-code in Listing 5.1, using ScaFi library functions `gradient`, `C`, and `broadcast`—cf. Section 3.3.2.3. Notice that since clusters are represented as aggregate processes, and aggregate processes define "scopes" for collective computations, the participation of an agent in an aggregate process has by itself the information about the cluster membership; so, collective tasks may be assigned to any cluster, and these will be inherently played

Listing 5.1: ScaFi pseudo-code of the clustering meta-algorithm

```
// process starts when r(i) is positive
val newProc = mux (r(i) > 0) { Set(mid) } { Set.empty }
// collect map from ℓ ∈ to (m_i^ℓ, u_i^ℓ)
val clusters = spawn(ℓ => _ => {
  val m_i^ℓ = gradient(mid == ℓ, d) // distance estimation
  val c_i^ℓ = m_i^ℓ < r(ℓ) // whether device is in cluster
  val t_i^ℓ = C(m_i^ℓ, f, s(i)) // summary collection
  val u_i^ℓ = broadcast(m_i^ℓ, t_i^ℓ) // summary broadcast
  return ((m_i^ℓ, u_i^ℓ), c_i^ℓ) // process result and status
}, newProc, ())
// selected leader
val l_i = mux (r(i) > 0) {
  clusters.filter(x => p(x._2, clusters(mid))).keys.min
} { mid }
// selected leader summary
val w_i = mux (r(i) > 0) { clusters(l_i)._2 } { None }
// propagate in process
val result = spawn(ℓ => _ => {
  val m_i^ℓ = clusters(ℓ)._1 // recover distances
  val c_i^ℓ = m_i^ℓ < r(ℓ) // whether device is in cluster
  val (l_i^ℓ, w_i^ℓ) = broadcast(m_i^ℓ, (l_i, w_i)) // final broadcast
  return ((l_i^ℓ, w_i^ℓ), c_i^ℓ) // process result and status
}, newProc, ())
// build result map
return result.map(x => { x._2._1 -> x._2._2 })
```

by all the members of that cluster. We also remark that although values v_i are not directly used by the meta-algorithms, the parameters $r(i)$ and $d(i, j)$ are allowed to depend on them (and usually do), so that values are indirectly used. An example of such behaviour is given in the next section.

5.4 Evaluation

In this section, we evaluate the meta-algorithm proposed in Section 5.3.4 in a case study of situation recognition within a synthetic environment (Section 5.4.1). The goal (Section 5.4.2) is to show how the algorithm can cluster agents in a sensing-based fashion, hence identifying various temperature cluster shapes. Furthermore, we assess how the algorithm works in mobile settings, where a swarm of agents moves across an environment—which can be representative for exploration scenarios. After describing the scenario and goals, in this section we describe the simulation framework (Section 5.4.3), the simulation configurations (Section 5.4.4), the results (Section 5.4.5), and finally provide a discussion about the evaluation and the approach (Section 5.4.6).

5.4.1 Scenario Description

A swarm group of *robots* is interested in identifying areas where environmental data varies within a known range. In particular, we assume that the robots are both capable of sensing the environmental temperature, perceiving their position in space (e.g., using GPS), and exploring a limited area (i.e., a square with a side of 1 km). The temperature is just an arbitrary choice of a sensible physical quantity that should drive, together with the spatial distribution, the clustering; the idea is that a temperature can be indicative for an environment situation that could require attention or intervention (cf. wildfires which can start and spread in hot, dry, and windy conditions). The scenarios are plausible, but we are not interested in full realism: simplifications and generalizations are introduced to study the algorithm in diverse controlled situations. Since the absence of central authority and the limited agent communication capability, we suppose that the agents can only interact with their neighbours (i.e., the devices with which a agent manages to establish a connection). In particular, we imagine that each agent is equipped with a LoRa module with a connection range of 100 m. In this case, a node can potentially participate in several clusters as it may be spatially close to two different phenomena. Therefore, it must both partake in the collective perception (i.e. perceive the local temperature) and act to solve the cluster-identified problem. The choice of how and when a node should act depends on the application but is typically left to the leader, since it has the cluster-side vision of phenomena and

the nodes. Notice that these assumptions are coherent with the system model of Section 3.3.1.

In the experiments described in the following, we are only interested in the clusters determined by the swarm cooperatively, not in *how* clusters are leveraged at the application level. However, even if we do not directly leverage the output of the clustering process, we would highlight that, in using the proposed algorithm, we inherently exploit *both* the leader election process and the multi-cluster formation. The foster is necessary to create clusters since, in our algorithm, each cluster is managed by *one* leader. The latter is essential to track the phenomena of interest. In fact, as phenomena can be spatially close and thus overlapping, if a node could only participate in one cluster, we would not be able to analyse the traced phenomenon correctly. Lastly, we emphasize that the scope of this application is quite versatile and can be adapted to various specific scenarios as outlined in previous research [Sch+20]. Examples include marine monitoring [Far+17] (covering aspects like aquaculture, pollution, and water quality), intelligent agriculture [Bal+13] (involving fertilization and pest control), military surveillance, tracking of criminal activity, and locating victims in disaster situations [Sae+10].

5.4.2 Evaluation Goals

We set up these simulations to:

1. verify the capability of the algorithm to find different cluster shapes: To assess the algorithm's versatility, we aim to confirm its robustness in accurately identifying various types of data distributions, including but not limited to Gaussian shapes;

2. examine how found clusters can cope with drone movement and failures: upon confirming the algorithm's effectiveness in stationary conditions, we intend to explore its responsiveness to variables such as drone *mobility* and system *failures*. Our analysis will focus on how these dynamics affect the clustering process in terms of cluster count, shape, and size;

3. test the algorithm dynamics when the temperature distribution changes: in a swarm agentics context, the observed phenomena could change over time. Therefore, the algorithm proposed should be robust against phenomena dynamisms.

That is, these goals reflect the design requirement of supporting sensing/spatial-based clustering in static, mobile, and environment-dynamic scenarios.

5.4.3 Simulation Framework

We verify our sensing-driven clustering algorithm using Alchemist simulations. The simulation experiments, resulting data, source code, and instructions for reproducibility are available at a public GitHub repository[4]. Alchemist is already used in similar scenarios [Cas+21], and it supports the ScaFi language [Cas+20a], that has been chosen among other field-based languages [Vir+19] as it supports aggregate processes [Cas+19], which we consider essential in order to implement our clustering algorithm.

5.4.3.1 Parameters

To evaluate the efficacy of our proposed solution, we examine the collective program behavior under various parameters. These parameters are summarized in Table 5.2 and elaborated upon below.

A key parameter is the *in-cluster threshold* (θ), which serves as a determinant for whether an agent resides within or outside a cluster. This threshold plays a critical role in guiding the expansion of the aggregate process across nodes. A low value may limit the program's consideration to only a handful of nodes, whereas an excessively high value could result in the inclusion of nodes that should not be part of the cluster. Given that this parameter is contingent on the specific application, developers need to judiciously strike a balance between node inclusion and boundedness, as this will significantly influence the shape of the cluster.

The *same cluster threshold* (γ) is employed by the cluster leader to determine the similarity between two clusters, as elaborated in Section 5.3.4. This parameter is vital in accurately establishing cluster boundaries. A high value for γ could result in the merging of two distinct clusters, while a value that's too low might leave overlapping clusters separated even when they could be meaningfully combined.

A clustering process starts when a node becomes a candidate. *waiting candidate time* (β) rules the rounds needed by a node to spawn a process after it has become a candidate. This helps in avoiding the excessive process spawn due to small local temperature variations.

We are interested in the robustness of the clustering process against the robots movement. Therefore, we tested our solution varying the robot *speed* (ω) and the *exploration range* (ζ). We expect that the higher the movement speed, the greater the instability of the identified clusters. ω does not affect candidate robots, they will stand still until they stay candidates.

We check also how the output changes varying the *density* (α) of robots. Theoretically, we expect a better result with high-density swarms. From α we compute

[4] `https://github.com/cric96/experiment-2021-swarm-intelligence-si`

Parameter	Unit	Description	Values
In Cluster Threshold – θ	°C	A real value used to verify if the temperature perceived in a certain node could be considered as a part of the current cluster	[0.5, 1.0, 1.5]
Same Cluster Threshold – γ	n.a	A real value used to verify if two clusters could be considered as the same	[0.1, 0.3, 0.7]
Speed – ω	km/s	The constant velocity used by drone to explore the areas	[7, 10, 14]
Exploration range – ζ	km	The maximum range area in which drones could move	[0.5, 0.6]
Density – α	n.a	A parameter used to define how many nodes will be placed in the environment	[0.5, 0.75]
Waiting candidate time – β	n.a	Rounds needed to mark a node as candidate	[3, 5, 7]
Failure frequency – ξ	Hz	Failure frequency of random nodes that participate in the system	[0.5, 0.1, 0]
Spawn frequency – τ	Hz	Spawn frequency of a node in a random position within the environment	[0.5, 0.1, 0]

Table 5.2: A summary of the parameters used in simulations

the total number of robots as: $N = (10/\alpha)^2$, e.g. with $\alpha = 0.5, N = 400$ and with $\alpha = 0.75, N = 173$.

Finally, ξ (*failure frequency*) and τ (*spawn frequency*) are used to verify how our algorithm could handle failures during the clustering process. The foster rules the frequency in which a random node disappears from the system. The latter controls the rate of spawning nodes that will participate in the aggregate program evaluation. This is useful to avoid complete node isolation after frequent node failures. Even if the movement is already a good estimation of how the system responds to dynamisms, we want to add another disruptive change. Indeed, movements are typically relative, and therefore the changes in the neighbourhood are limited.

5.4.3.2 Metrics

The clustering results are verified using different metrics. First, we extract the number of total unique clusters found by the collective to check if the program produces the correct partitioning. This value gives a quick overview of the clustering result. Along with this value, we evaluate the total number of unique merged clusters. The latter should be as near as possible to the correct cluster number.

However, neither the number of total unique clusters nor the total number of unique merged clusters tells us anything about the shape of the clusters. To this aim, we compute several metrics:

- the number of nodes for each cluster, stating the overall device partitions;
- the Silhouette [Rou87] and Dunn [Dun74] indexes, used as internal evaluation schemes;
- the error rate, observable only when we know the ground truth.

By examining the Silhouette index, we can gauge the extent to which the clusters overlap. A Silhouette value approaching 0 indicates overlapping clusters, while a value closer to 1 suggests that the clusters are distinct and well-separated. The Dunn index serves as a supplementary metric; when the Silhouette index is close to 1, a higher Dunn index value is expected. The error rate metric quantifies the degree of node misclassification. A node is considered misclassified if it is either falsely associated with a cluster (false positive), or if it is incorrectly identified as external when it should actually belong to a cluster (false negative). The error rate is calculated using the following formula:

$$E = \frac{FP + FN}{TP + TN}$$

Here, TP denotes *true positives*, which refers to the number of nodes correctly classified within a cluster and located near a specific variable like temperature distribution. TN represents *true negatives*, or the number of nodes accurately classified as external and distanced from any variable distribution. This metric provides insights into the algorithm's performance during drone exploration.

5.4.4 Simulations

We evaluate the behaviour of our algorithm in several experiments. The simulations have in common 1. the environment area (a square with a side of 1km), 2. the communication radius (100 m), and 3. the average evaluation frequency of aggregate programs (1 Hz). The drones are uniformly placed to cover the entire zone.

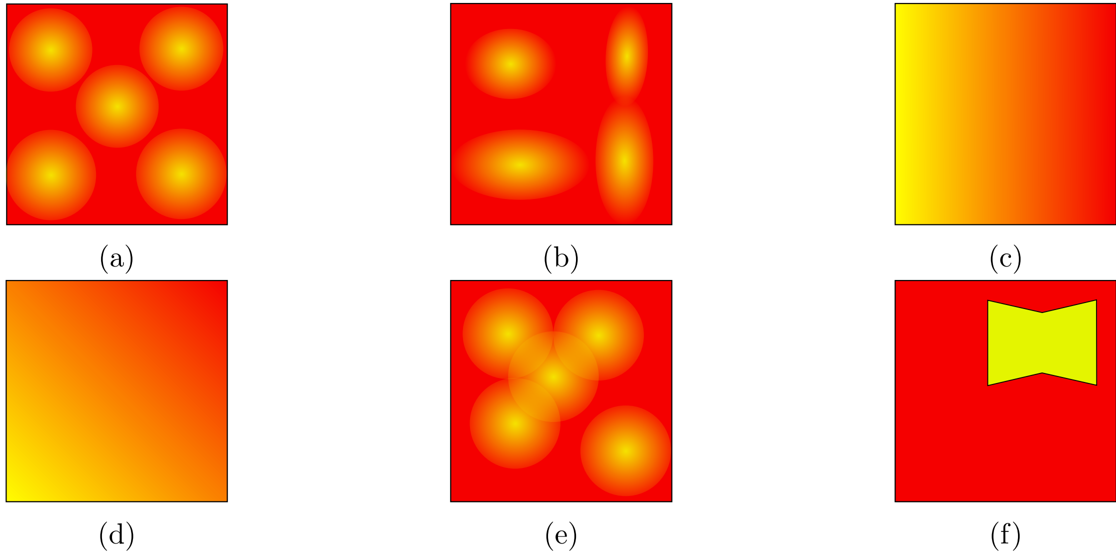


Figure 5.2: Graphical representation of temperature field distributions used in the simulations. The lighter the colour, the lower the temperature.

We run the simulations in a modern machine equipped with two AMD EPYC 7301 with 128 GB RAM. The results are reproducible in any modern machine, but consider that it might take a long time to finish (in our configuration, the simulations end after 8 h). Each scenario is executed 20 times with different random seeds for a total of 100 simulated seconds (some simulations lasts 150 s to reach convergence). The selection of scenarios presented below aims to achieve two main objectives:

1. assess the effectiveness of our algorithm,
2. ensure that it meets all the previously outlined goals.

Specifically, we chose scenarios where most temperature distributions follow a normal distribution. This choice is grounded in the observation that natural phenomena often exhibit such distributions. Consequently, if our algorithm succeeds in identifying clusters in these Gaussian scenarios, it is likely to be effective in other contexts where monitoring natural phenomena is essential. To complement this, we also tested the algorithm's ability to identify non-Gaussian shapes, as evidenced in scenarios 3, 4, and 5.

The final set of scenarios is designed to evaluate the system's adaptability to changes at both the system level, such as movement and failures, and the environmental level, like variations in distribution over time.

The simulation data is processed using NumPy [Har+20] and visualized through matplotlib [Hun07]. The plotted results consist of the average (lines)

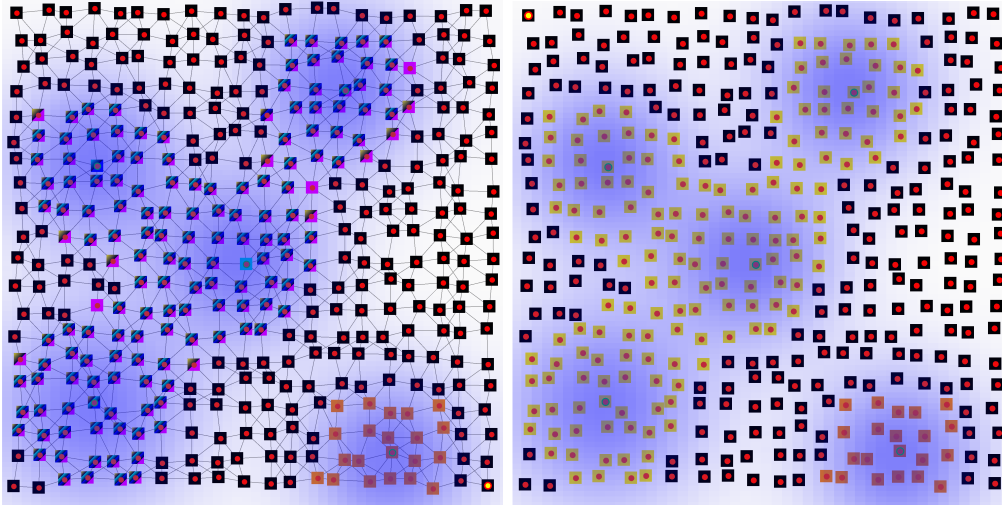


Figure 5.3: Snapshots of simulation executions. The colour of the square identifies the cluster ID found in that point. Black colour means no cluster. The green circle means that the node is a candidate. The blue gradient circles are a graphical representation of temperature distribution. On the left is shown a snapshot of a simulation before the merge policy has been applied (multiple clusters per point are found). On the right, there is the snapshot of the same simulation after the merge policy action.

and the standard deviation (area behind lines) of the values of interest in different episodes. In Figure 5.3 there is a graphical representation of a run of our algorithm.

5.4.4.1 Scenario 1: Gaussian patterns (Figure 5.2a)

Description In this scenario, the drones are stationary (i.e., they stand still). There are five zones with a Gaussian distribution, and there is no overlap between distributions. Given the stationary situation, the number of candidate nodes is equal to the number of zones of interest.

Why Used to verify 1, particularly we expect that the algorithm finds clusters without making any errors and that they will be stable over time.

5.4.4.2 Scenario 2: Stretched Gaussian patterns (Figure 5.2b)

Description These simulations are similar to the previous one, but in this case, the Gaussian distributions have an ellipse-like shape.

Why With these experiments, we would check that the shape does not make such a difference in the clustering process. Indeed, we expect a result similar to the one in the previous example (1).

5.4.4.3 Scenario 3: One direction temperature field (Figures 5.2c and 5.2d)

Description In this case, we imagine that only one cluster is present (fixing θ to 1 °C and putting a total variation of temperature equal to 1 °C). Temperatures grow from left to right in a constant fashion. Namely, in Figure 5.2c the temperature varies in one dimension (horizontally), whereas in Figure 5.2d the temperature varies in two dimensions (diagonally). In the scenario depicted in Figure 5.2c we are interested to see what happens when multiple candidates are elected. In this case, there are several relative minima (the set of nodes that are leftmost with minimum ID in their neighbourhood). But, eventually, the processes will expand them in the same way. Thus, we expect that the merging policy tends to create only one cluster. We use the scenario shown in Figure 5.2d as a reference. Indeed, there will be only one candidate (located in the bottom left corner), and hence the algorithm should result in one cluster.

Why We devise these experiments to test the effectiveness of the merging policy and to verify the goal 1.

5.4.4.4 Scenario 4: Gaussian overlapped patterns (Figure 5.2e)

Description In this case, we have several Gaussian patterns that could be overlapped. We imagine that the θ value is essential here: if the value is too high, the system will recognise the set of overlapping clusters as one; otherwise, it will consider disjointed.

Why This experiment serves to emphasize that θ is a domain-dependent choice. Moreover, it will show that the algorithm could be used also to find overlapped situations (1).

5.4.4.5 Scenario 5: Non convex patterns (Figure 5.2f)

Description In this case, there are two zones, one with a non-convex shape with a lower temperature than the outer zone. Here we expect that, eventually, the system will identify the presence of only two clusters. The program might identify several candidates in the transitory phases (cf. one for each edge). Hence, the merging policy should fix this issue by producing only two clusters.

Why With this scenario, we want to point out that the program can cope with zones of arbitrary shape.

5.4.4.6 Scenario 6: Gaussian patterns with movement

Description We test the result using four Gaussian distributions (arranged similarly to Figure 5.2a) combined with movement. Here, both merging policy, and *waiting candidate time* (β) will be essential. In particular, β helps to avoid false positives since it waits before spawning a new clustering process when encounters small local temperature variations. In general, we imagine that high values of ω and ζ will make the algorithm more unstable.

Why We are interested in seeing how movement affects the result of the clustering process (2).

5.4.4.7 Scenario 7: Variable size Gaussian pattern

Description In this experiment, the temperature distributions are placed similarly as Figure 5.2a, but then the size of areas evolves in time. We expand the areas until a time T and then contract them to their initial size. The starting area range is 100 m, and the maximum area expansion is 1 km. Here we expect that the cluster area follows the underlying temperature distribution.

Why In this experiment, we verify the algorithm's robustness against temperature changes (3).

5.4.4.8 Scenario 8: Random Failures

Description The temperature distribution of choice follows the Figure 5.2a. Nodes could disappear randomly with a rate specified by *failure frequency*. This could be harmful when: i) the failure happens in a leader node, and therefore the cluster formed should be destroyed and, ii) the failures are so frequent that certain nodes became isolated. The second case is avoided using *spawn frequency*, which forces the system to insert a new node with the specified rate. In this case, we expect robust performance with high-density system (i.e., $\alpha = 0.5$) since spurious failure does not change the overall topology.

Why In this last scenario, we check how the system handles node failures during the clustering process (3).

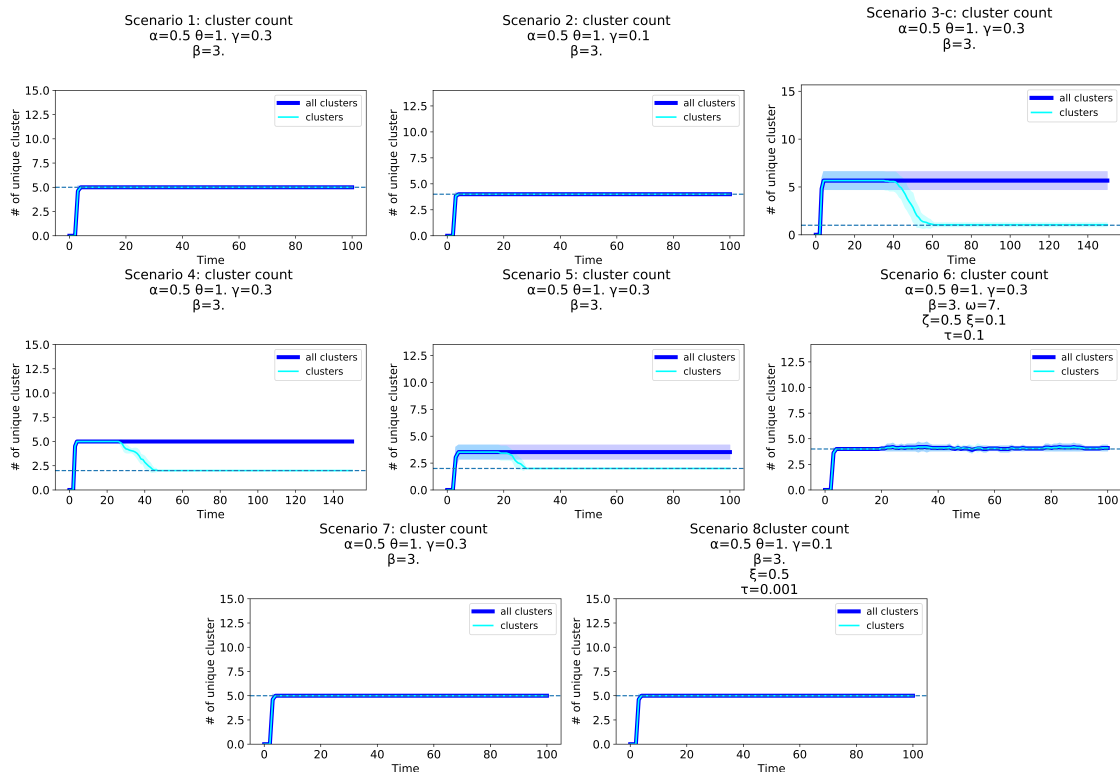


Figure 5.4: Overview of simulation results. The dotted lines identify the ideal cluster division count. The blue lines show the unique cluster found. Instead, the cyan lines indicate the unique cluster number after the merging phase.

5.4.5 Results

The simulation results underscore the algorithm's capability to effectively partition the data into meaningful clusters. As demonstrated in Figure 5.4, our algorithm stabilizes to produce the correct number of clusters after a certain settling period. In the sections that follow, we concentrate on discussing the outcomes in relation to the evaluation goals outlined in Section 5.4.2.

Goal 1: static sensing/spatial-based clustering Running the simulations of scenarios 1-5 we verified how much the clusters extracted follow the underlying temperature distribution in the static context. Figure 5.4 shows that the algorithm correctly extracts the cluster number – with the optimal parameters' configuration. Furthermore, observing Figure 5.5, we can deduce that the cluster shape is correct too. Indeed, the Silhouette index tends to be 1 when the clusters are disjointed, and the error rate is negligible.

Here, θ plays a key role. Observing the behaviour of scenario 4 in Figure 5.6, we

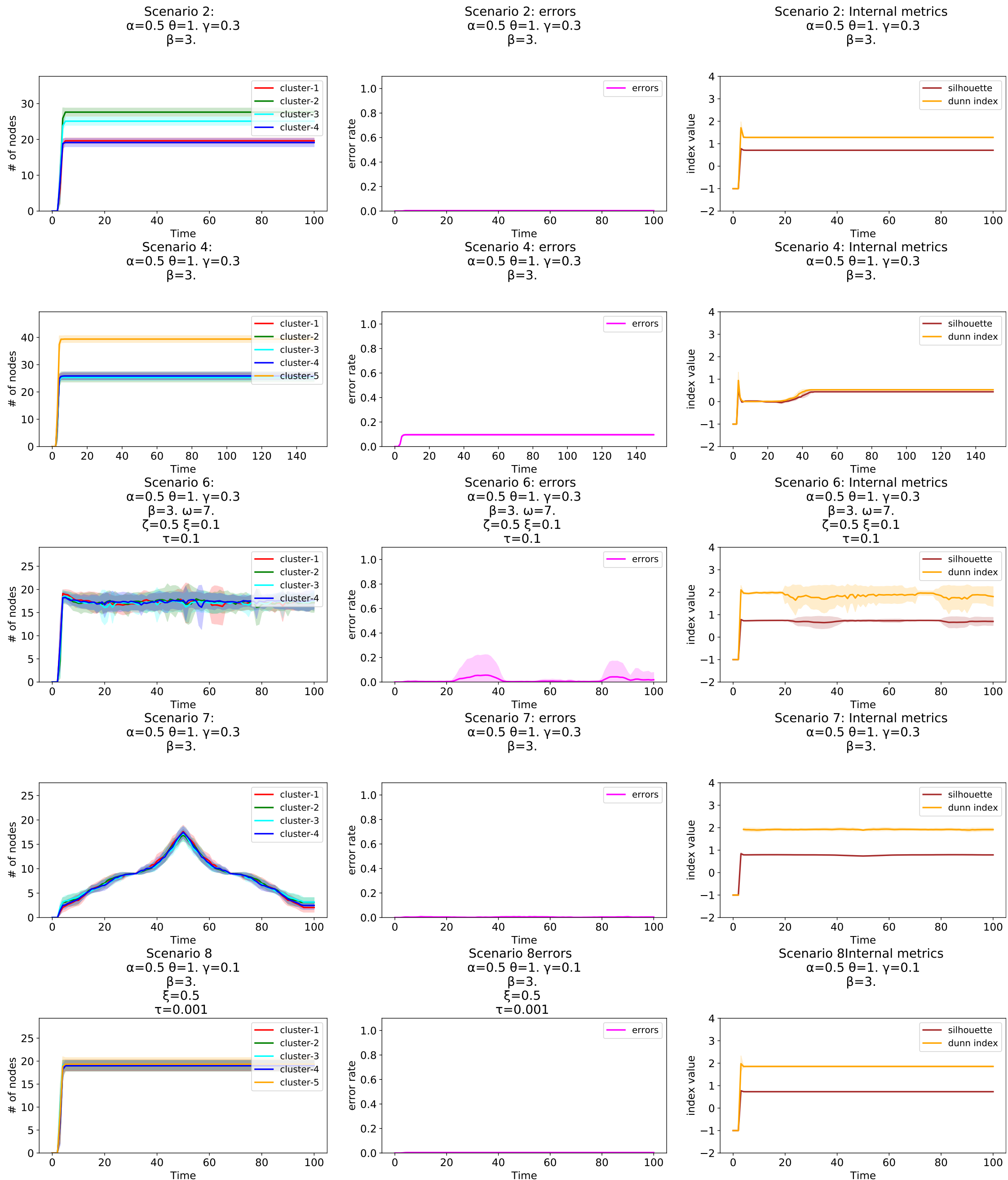


Figure 5.5: In-depth analysis of good simulation results. In general, the algorithm produces good results. In the case of movement and failures, the error can reach up to 10 per cent.

see that with too low θ we overestimate the cluster numbers and, with a high level

of θ, we underestimate the cluster number. But this was the expected behaviour, as it depends directly on the trend of the target distributions.

Finally, Another important aspect is the density (α) of the system. With a few nodes, candidate nodes may be positioned far from the cluster centre, thus identifying wider areas than expected.

Goal 2: robustness against node mobility *and failures* When nodes have a low mobility and exploration range, the system is robust to node movements (Figure 5.5). The exploring policy introduces errors, but the results are comparable to solutions where the nodes are stationary. Moreover, even in case of failures, the clustering process is practically not affected at all. However, in the worst case, mobility and failures lead to false positives (Figure 5.6). Indeed, some processes start in areas where the temperature is almost constant. Therefore, that process approximately covers the whole area (and hence produces a high error rate). Scenario 8 is mainly influenced by the low-density situation. Indeed, in that case, removing nodes lead to not covering the whole system.

Goal 3: robustness against temperature changes The result of scenario 7 is comparable to the static scenario. Indeed, Figure 5.5 shows that the cluster number is correct, and Figure 5.5 shows that the error rate is low, and the shape is accurate. The solution suffers from low-density values and wrong θ values as scenarios 1-5 (Figure 5.6).

5.4.6 Discussion

Simulations

Ultimately, our algorithm can support a certain degree of movement, sporadic failures, find various cluster shapes, and cope with temperature changes in the optimal condition: high density (α), limited exploration range (ζ), and an appropriate value for in cluster threshold (θ) value.

However, when drones move randomly, the algorithm starts to produce sub-optimal cluster divisions since the nodes do not care about the cluster found, and they continue to explore the area. But this could lead to becoming a false candidate and then starting an unwanted clustering process. Furthermore, it could be argued that uniform zones are part of a cluster that is not identified as there are no relative minima. For this reason, when a node starts the process in a non-correct zone, the cluster identification will expand in the nearly whole system. This problem could be reduced by changing ω and ζ when the nodes belong to a cluster.

It is important to highlight that when α is set to a low value, the algorithm tends to produce poor cluster divisions, a phenomenon that becomes particularly

evident in the presence of failures. This limitation is inherent to our algorithm's design, which relies on a centroid to initiate the clustering process. With a low α value, the likelihood increases that the node initiating the clustering process is distant from the true centroid of the cluster. As a result, the expansion process may deviate from the underlying distribution, leading to a significant misclassification of nodes.

Hardware Deployment

While we have not conducted experiments involving the clustering algorithm on a physical system, insights can be gleaned from the physical deployment of FCPP [Aud20], a C++ library that provides an internal DSL for field-based programming. This deployment was carried out in the context of an Industrial Internet of Things (IIoT) scenario [Tes+22]. The hardware used consisted of DWM1001C modules by Decawave, which are resource-constrained with a 64MHz ARM Cortex-M4 CPU, 512 KB of flash memory, and 64 KB of RAM. Despite these limitations, the porting of FCPP was successful, and we were able to execute a field-based program with dynamic processes whose complexity is comparable to the clustering algorithm discussed in this paper [Tes+22].

The DWM1001C modules support BLE (Bluetooth Low Energy) and UWB (Ultra Wideband) for communication. In the IIoT context, BLE was used for message exchange among neighboring nodes, while UWB was employed for distance estimation. This distance data could be integrated into the current clustering algorithm using the `gradient` function for multi-hop distance estimation.

Although the FCPP deployment experience has been promising, pointing toward the feasibility of deploying our clustering algorithm in a similar setting, several differences between the two scenarios merit further investigation. Firstly, the largest IIoT experiment involved only 20 nodes, which may be insufficient to adequately assess the clustering algorithm

5.5 Related Work

Coverage of related work is organized to separately cover: related swarm-based environment monitoring approaches (Section 5.5.1), related clustering models and problems (Section 5.5.2), research work related to the sensing-based clustering problem we address (Section 5.5.3), research work related to field-based computing (Section 5.5.4), and related field-based algorithms (Section 5.5.5).

5.5.1 Swarm-based Environment Monitoring

The approach proposed can be used to dynamically cluster a swarm, e.g., to monitor an environment in a decentralized way. Literature on swarm-based environment monitoring is ample [DM12]. In particular, various works leverage mobility and sparse sampling [Bes+19; Cas+22b; Kem+17].

In [GA14], a persistent monitoring approach of environment phenomena with discontinuous dynamics is proposed. It is based on optimally adapting a sparse set of sensing locations according to an evolving stochastic model of the environment. In [Bes+19], decentralized planning is used to support multi-agent active perception, which leverages movement to improve the quality of information gathering through effective choice of "viewpoints" in space and time. In [Kem+17], the authors focus on multi-agent coordination for informative adaptive sampling in unknown, communication-constrained environments (like lakes or oceans). Their approach is based on dynamic, decentralized Voronoi partitioning over a set of sampling locations, which are recalculated at synchronization points initiated through requests for surfacing events. Though the approach of this paper could also be used to support *sparse sampling* [Cas+22b], it also aims at supporting the formation of spatially cohesive clusters for coordinated processing and/or action. Moreover, we do not aim at moving agents to appropriate sampling locations, but rather leave the agents to move autonomously (e.g., according to exploration policies) while having the collective clustering reflect the underlying phenomenon to support decision-making possibly beyond pure environmental sampling. The use of Voronoi partitions in [Kem+17] differs from our clustering in that they leverage regions to limit the prospective sampling locations to be visited by each vehicle, while we actually want to define *groups* of coordinating agents.

5.5.2 Related Clustering Models and Problems

Clustering is a well-known problem in data analysis and machine learning, and has been widely studied in the literature [JMF99; Est02; Jai10]. In a classical setting, the data to be clustered is stored in a single dataset, and a single algorithm (or agent) is in charge of finding the "best" clusters according to some optimization criteria. Each data point in the input data set is described by the values of a fixed set of *features*; the number of such features constitutes the dimensionality of the data set and, typically, high dimensional data is harder to cluster meaningfully.

A characteristic of the clustering tasks considered in the present paper (and in general, of sensing-based methods, see the next section), is that besides the sensed data, a main source of information is the spatial distance between the agents. In [TU21], the authors consider high-dimensional data sets that exhibit *natural clusters*, characterized by distances and/or density-based structures. They propose

a semi-automated method whereby the clusters are automatically proposed and manually selected starting from a topographic visualization of the high-dimensional data. Notably, they use swarm intelligence for computing the topographic map, while other techniques are adopted for the interactive process of clusters computation.

There are, however, several works that address swarm-based clustering, using swarm intelligence for the clustering task itself [MBF11]. It is important to note that such methods (both those based on particle swarm optimization (PSO), and those based on ant colony systems (ACS)) exploit swarms just as a computational means for finding clusters in a data set. Their goal is not to cluster the elements of the swarm itself, as it is the case for the present work, but to simulate a virtual swarm to find good quality clusters.

Some works directly address the clustering of swarms. In [Hu+21], the clustering of a team of special agents (i.e., aerial drones) is part of a larger process that, after cluster formation, also involves formation tracking (i.e., tracking a target through a suitable formation), and containment control (i.e., surround ground agents cooperating in the mission). The method proposed to form clusters is based on a *game-theoretic* framework named GRAPE. A significant difference w.r.t. the present work is that the number (and nature) of clusters is determined by a given set of targets, while we do not assume such a priori knowledge. Another significant work with similar goals is [GHZ18], where a team of agents must be partitioned into clusters organized as suitable *formations* (i.e., geometric spatial patterns). The proposed solution inter-mixes the determination of clusters and their formation (based, among other things, on the agents' dynamics), assuming that the number and nature of such formations is known a priori.

Since we consider clustering over a given topology (network), the problem can be related to graph-based clustering [CJ10]. Graph-based clustering, however, assumes that the given graph can be partitioned into densely connected subgraphs that are sparsely connected to each other; i.e., it assumes that all the similarity information is expressed by the presence of edges between nodes (and, possibly, by their weights). This is not necessarily the case with the networks formed by our swarms, where connections are just determined by spatial distance, and the clustering is strongly influenced by the sensed data. Also, community detection methods can be viewed as clustering of the nodes of a graph representing a network of relations (e.g. a social network) [Jav+18]. Interestingly, unlike in generic graph-based clustering, communities can easily overlap, since a node (e.g., user) may belong to several communities at once.

5.5.3 Related Work on Sensing-based Clustering

Sensing-based clustering typically applies to sensor networks that are distributed on a geographical area and exploit clustering mainly to reduce the communication bandwidth and/or energy consumption of the net. The role played by sensing a (possibly dynamic) geographic environment makes such problem and the proposed approaches to solve it relevant to the present work, although the agents considered here are themselves dynamic entities moving and acting across the space.

In [LM07], the goal is to partition sensors for indoor monitoring and control. The cluster heads are predetermined (based on the sources to be monitored and controlled), while cluster formation is periodically scheduled in order to adapt to changes in the sensed data. In our work, instead, the cluster heads are not a priori given: they are determined according to the *sensed data* (e.g., the agents perceiving local minima) and can change *dynamically* (e.g., because a candidate withdraws and joins a different cluster).

The goal of [GLY07] is, instead, to obtain energy savings in data collection from a wireless sensor network by receiving values from only a subset of selected representatives and predicting the other valuer through automatically generated statistical models. Cluster heads are chosen (probabilistically) based on the amount of energy they have. Cluster formation is periodically scheduled, and the assignment of a sensor to a cluster is based on the distance from the head and the similarity of the sensed value with the head's value. A work with similar goals is [CZ18], where again energy savings in a WSN is the primary motivation. Here, the cluster heads are chosen based on residual energy level and data gradient. Moreover, an autoregressive prediction model for sensory data is maintained by each head to self-adjust temporal sampling intervals within the cluster.

A sensing-based clustering problem is also studied in [Kuc+20] where, however, instead of being a high energy-constrained WSN, the deployed system involves sensorized units and mobile phones able to upload all the relevant data to the cloud, through cellular and Wi-Fi connections. In a disaster scenario, the mobile phones data is used to centrally compute density-based clusters that can inform the SAR (Search and Rescue) teams about the location of people in the area.

The *DyClee* approach described in [RTG19] is also centralized. The authors assume that streams of sensors observations (e.g. in an Industrial IoT) are continually tracked by their system, and are classified (e.g., as healthy or faulty) based on a set of clusters that capture the patterns corresponding to different states. The main focus is on the novelty detection problem, or concept drift, which implies the ability to update the clusters as new behaviour is learned, while ignoring noise and occasional outliers. The online clustering algorithm consists of two stages based, respectively, on distance and density, and is *fully dynamic* in that it is able to create, eliminate, drift, merge, and split clusters as data is processed.

5.5.4 Related Approaches and Programming Models

Programming swarms of agents is a difficult task, because of the need of coordinating their local behaviours to achieve global, swarm-level goals. In this work, we adopt the field-based computing and programming approach [Vir+19] for expressing self-organising, collective behaviour of swarms. Our focus is on decentralized behaviour-based approaches (rather than automatic design methods like e.g. reinforcement learning), as surveyed e.g. in [Bra+13; Vir+19] and briefly in the following.

An approach to the problem that has proven to be quite effective is generative communication through tuple-based coordination models, as offered, e.g., in the Linda language [Gel85] and its descendants; essentially, several processes running on the same system can synchronize by writing and retrieving information in a shared (tuple-)space. A derived idea is that of allowing programmability of the tuple space itself, so that the coordination logic of processes can be embedded in the communication medium–see, e.g., [OD01]. An obvious limitation of the mentioned approaches for the task of swarm programming is that they assume a central memory accessible by all the agents/processes. However, the idea of tuple-spaces has been extended also to distributed systems, e.g., in the IBM TSpaces framework [Wyc+98].

An important feature of swarm systems is their adaptivity achieved through self-organization. A support to build such kind of systems is offered by frameworks inspired by other sciences such as biology [TM03] and chemistry [Say09]. The field-based computing approach adopted in the present paper is based, instead, on the concept of *field*, borrowed from physics. The related idea of a *field of tuples* has been implemented in the TOTA middleware [MZ09].

As seen in this chapter, the field-based approach is particularly well suited to mobile, spatially situated agents. A related (and precursor) thread of research of that of *spatial computing*, where space is both an abstraction and a means for computation. Spatial computing approaches have been largely surveyed in [Bea+13]. They are also related to *macro-programming* [NMW07], where distributed systems as wholes are programmed by a centralized perspective. For instance, a prominent related macro approach to swarm programming is Buzz [PB16a], where swarms are first-class collection-like abstractions.

5.5.5 Related Field-based Algorithms

Field-based computing has the peculiar ability to capture collective behaviours as functions operating on fields and to compose them together as "building blocks" to address problems of increasing complexity [Vir+19]. Of particular relevance for the present discussion is the implementation of the *SCR (Self-Organising Coordination*

Regions) pattern. Most specifically, the SCR pattern can be denoted as a feedback chain S-G-C-G: leaders are elected (S); then, a gradient from leaders builds the communication structure (G); then, data from members (indirectly defined by the information path towards a leader) is collected towards leaders (C); then, data from leaders is propagated back to the members of the regions (G). However, the SCR pattern is not limited to clustering (S-G part), but also regulates interactions within regions (C-G part). Roughly, the sensing-based clustering algorithm covered in this work could replace the initial C-G composition that determines the system regions.

Similar to a clustering algorithm, the S block [MBD18] provides a distributed mechanism to elect leaders from a set of candidates, and to assign each remaining *user* node to a leader, thus partitioning the system into regions. The approach presented here is different in several respects: first, the candidate leaders are determined by a characteristic of a *sensed* measure (e.g., local minimum); second, each candidate cluster head spawns an aggregate process to recruit other nodes within the cluster; finally, the other nodes can join more than one cluster, based on the similarity of their sensed values with the ones sensed by leaders.

5.6 Final Remarks

In this chapter, we precisely define and address the dynamic sensing-based mobile swarm clustering problem—an essential task in the context of CPSW, to support the coordination of collective tasks based on environmental sensing data. Most specifically, we use our language-based approach centred on aggregate computing to devise a novel configurable meta-algorithm promoting self-organised clustering in a swarm of neighbouring-interacting agents. The algorithm is evaluated on a set of synthetic environment configurations in the context of swarm robotics—a typical application domain for CPSW. In particular, we show that a swarm can autonomously create clusters reflecting the underlying dynamics of the perceptible target phenomenon in the environment, and can deal with changes in the swarm topology and environment. The subsequent chapter will introduce a general programming pattern for distributed sensing and actuation in CPSW systems. This pattern aims to provide a comprehensive solution for addressing the challenges associated with CPSW programming.

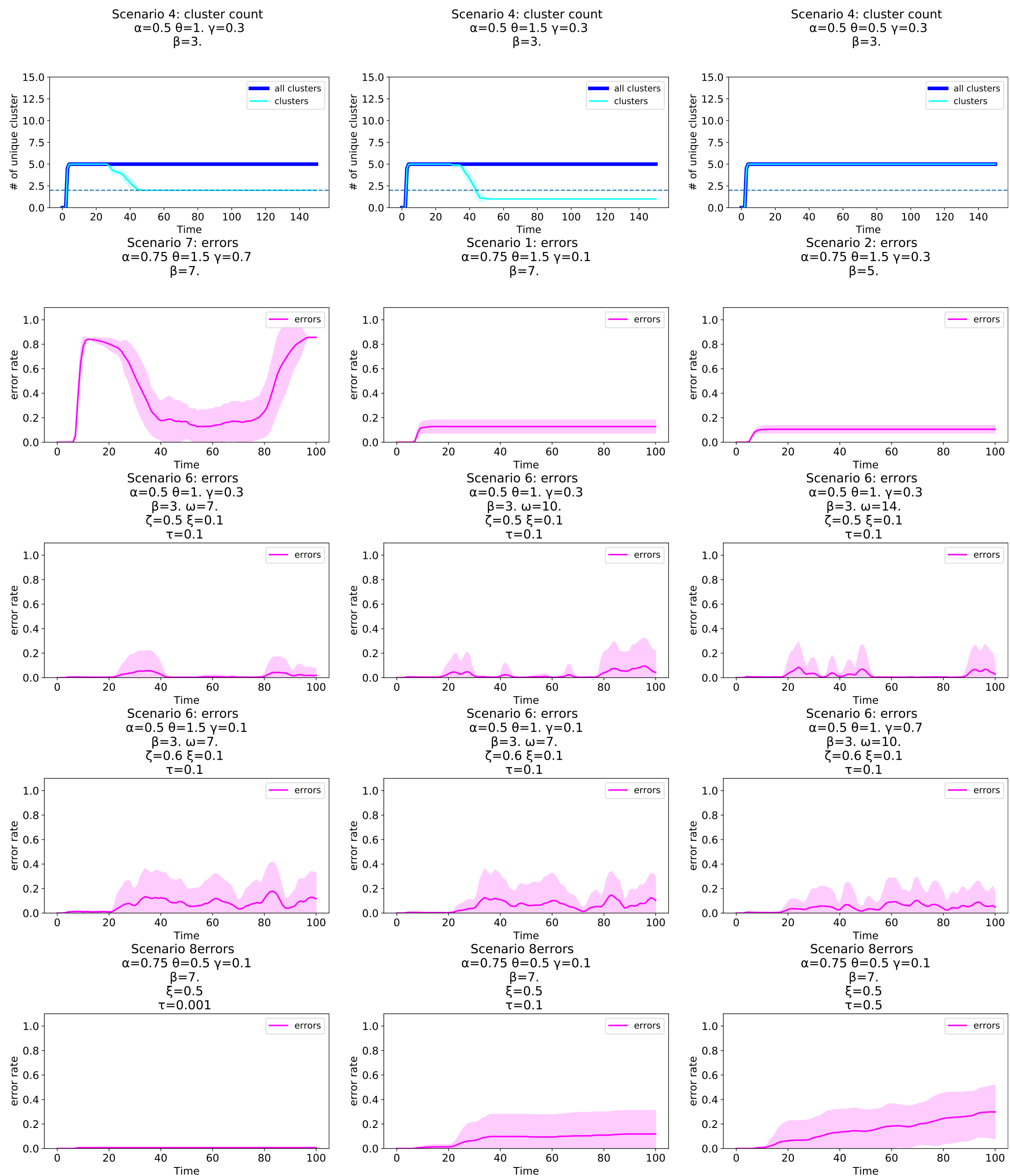


Figure 5.6: Main examples of bad clustering results. In the first line, the images show different behaviour varying θ. In the second line, the plots show how the algorithm does not handle well low-density robot swarms. In the third line, the charts show how the algorithm handles various movement speeds. The fourth line shows how the exploration range impacts the clustering results. Finally, the last line shows how failures impact performance.

Chapter 6

Patterns: Dynamic Decentralization Domains for Cyber-Physical Swarm

What is the right level of abstraction for programming CPSWs?
How distributed sensing and actuation can be organized to adapt to the environment?
– **RQ3, RQ4**

Contents

6.1 Motivation **122**
6.2 Decentralized situation recognition and action: a case study **123**
6.2.1 Requirements and abstractions 124
6.3 Dynamic Decentralization Domains in Practice **126**
6.4 Evaluation **127**
6.4.1 Experimental setup 129
6.4.2 Results and discussion 129
6.5 Final Remarks **130**

Edge computing and related scenarios like CPSWs promote a vision of distributed computational systems deeply integrated with humans and environments. The complexity and volume in terms of devices, communications, failures, and change,

are pushing the adoption of paradigms that can adequately address both functional and non-functional of CPSW that concerns of:

- *decentralization* for scalability and delegation;
- *autonomic computing* and *self-organization* [KC03] for operational effectiveness and adaptation;
- *in-network processing* for latency reduction and infrastructural autonomy; and
- *collective computing* [Cas+21] for coordination and collaboration.

Specifically, they form the platform upon which several concurrent *distributed computational processes (DCPs)* would run, carrying on transient activities by self-organized continuous computation and communication. The goal of a DCP is to identify dynamic regions of the computational environment (regions of "space") where situations of interest occur, monitor their evolution, and reactively trigger distributed actions to signal events, remedy problems, or control the phenomenon. Hence, DCPs are generated to satisfy a request, handle an event, or execute a collective task; they opportunistically spread (resp. shrinks) to gather (resp. release) resources/workers or cover (resp. uncover) regions of interest; they may perform distributed sensing and actuation; eventually, they may vanish once the activity is done.

In this chapter, we address the problem of capturing the right abstractions for modelling DCPs, abstracting from the specific communication technologies, taking inspiration from the analogous the approach taken in map-reduce frameworks for big data, where the declarative concept of *stream* is adopted. and propose the concepts of *concurrent collective tasks* and *decentralization domains*, which can be exploited in combination to provide distributed situated recognition and action.

6.1 Motivation

CPSWs are increasingly tasked with monitoring and acting upon dynamically changing environments, often without the availability of a central coordinator.

The recurrent approach in computer/software engineering to manage such complexity is to adopt various *levels* of abstractions and mechanisms that encapsulate coherent sets of problems and solutions. This work aims to simplify the programming of these complex systems by enabling the user to express high-level goals, without having to fully specify the *how*. This allows lower-level components to handle issues such as dynamicity, failure, and heterogeneity.

As an analogy, consider database management systems: SQL queries express what data needs to be retrieved, while the system itself determines the most efficient way to fulfill the request. The ultimate objective is to apply this principle to self-organizing systems, primarily to realize decentralized situation recognition and action.

6.2 Decentralized situation recognition and action: a case study

A CPSW system should ideally determine autonomously *what* has to be done, *when*, *where*, by *whom*, and *how*. The critical problem is setting up a *decentralized process for adaptive situation recognition and situated action*. The system should organize to monitor the environment for situations requiring intervention; then, the intervention should pursue the desired state of affairs. These two phases do not need to be sequential but can be performed continuously in a feedback loop, gradually steering the system towards a correct and stable configuration. Also, the system should *opportunistically* exploit available resources accordingly to the current context and goals—which may change dynamically. Also, we cannot assume the existence of a centralized coordinator such as the cloud, which is usually relied upon in classic approaches.

As an example and case study throughout the paper, consider a large-scale flood warning system, which we call FloodWatch, fully developed (in simulation) in Section 6.4. We want to monitor the rain intensity to pre-alert the public safety organizations close to areas at a *risk* of floods. The tracked phenomenon is spatially and temporally hard to predict with a fine-enough grain (data from the NOAA[1] has, at best, zip-code granularity): at a single-city level, we could perform better by promptly reacting to specialized sensor readings. However, the information provided by individual sensors is too fragile, as the risk depends on the rain intensity in the surroundings and not just on the specific spot (e.g., coastal zones with steep elevation profiles could suffer floods even with light rain, if the close-by higher-altitude zone is being hit hard). Pre-defining areas (using pre-existing altimetric and structural knowledge) helps, but this strategy misses out on essential information: how the underlying phenomenon is behaving. Indeed, areas should be formed ad-hoc considering the city structure and rain distribution, and leveraged to perform on-the-fly situation recognition and response.

This approach is practical whenever there are phenomena with non-strictly-local effects, irregularly shaped in space, and/or hard-to-predict at a fine grain.

[1]https://www.noaa.gov/

6.2.1 Requirements and abstractions

Given the high-level vision and goals discussed in the previous sections, and with the help of FloodWatch, we delineate some *needs* together with *abstractions* and corresponding *requirements*, for a programming model aimed at decentralized situation recognition and action.

6.2.1.1 R1. Concurrent collective task execution.

In FloodWatch, there is the need to coordinate a system that spans large geographical areas, hence leveraging DCPs for sensing, computation, and actuation at a collective level. One may also devise a more complex case study and platform for environmental monitoring where there is a distributed process for FloodWatch, another process for critical infrastructure monitoring, waste management, surveillance, etc.; these processes may run independently or, possibly, interact.

In general, most complex systems is not limited to a single activity but usually involve several activities running concurrently. Furthermore, these activities could be *collective*, i.e., involve a collaboration of multiple agents with partial perception of the environment. We call these *concurrent collective tasks (CCTs)*, which express activities that may *overlap* in the system (a device may partake in multiple CCTs simultaneously). Notice that CCTs may have a limited and dynamic domain: a subset of devices in the system (sometimes also called *team* or *ensemble*) which may change over time.

6.2.1.2 R2. Flexible and adaptive decentralization.

FloodWatch is centred on organizing distributed sensing and actuation according to both the environment structure and the current rainfall. Generally speaking, strategies that are too fine-grained or too coarse-grained tend to be sub-optimal: in the former case, non-local information is not considered, possibly resulting in a lack of coherence and global inefficiency; in the latter case, the system may fail to adequately recognize specific contexts that should be handled ad-hoc. In FloodWatch, warnings should be delivered in the surroundings of risky areas, but not too broadly.

Many systems, indeed [Pia+21a], often need abstractions capturing an "adaptive" *spatial divide-and-conquer* principle through which a problem in space is split into parts (or regions) that opportunistically adapt according to the context. We call each region a *decentralization domain (DD)* since it represents a non-overlapping bounded subsystem of a CCT. Multiple DDs can also *compete* to gather resources exclusively within the domain defined by a CCT (at whose level cross-domain interaction could happen, instead).

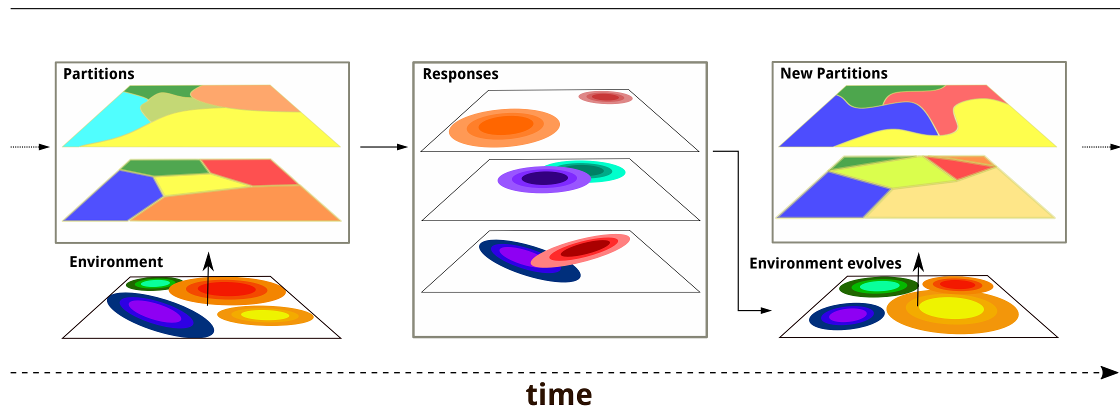


Figure 6.1: Overview of the proposed approach. The projected squares represent environments, with colours denoting environmental phenomena. Time flows left to right. The proposed system tracks spatial phenomena by partitioning the space in non-overlapping regions that agree on a measure, which is then leveraged to enact (multiple) spatially-bound responses that can overlap or compete. Contextual (induced or natural) changes are tracked by evolving (reshaping, deleting, or creating) the partitions.

6.2.1.3 R3. Feedback-regulated activity within decentralized domains.

In FloodWatch, each region should sense the water level and altitude, process data, and decide region-local actions such as alerting. In a system for computational resource management, each region might collect resource advertisements and requests, compute assignments, and publish assignments while also monitoring and handling the activity progress.

In general, DDs are expected to autonomously carry out distributed sensing activities, followed by processing and decision-making, which may trigger actions affecting the environment (cf. actuations, for a kind of indirect feedback) or spawning new CCTs (e.g., to connect with other services). Often, in order to simplify and guarantee consistency at the level of DDs, decisions might be taken centrally in a DD by a leader. This could also be seen as a feedback loop: contributions are collected at the leader, the leader performs the decision-making and spreads the decisions which are then carried out, producing new data to be gathered and so on.

6.2.1.4 Summary of requirements.

The rationale of the above requirements is to promote abstractions supporting concurrent, system-spanning, and possibly overlapping activities (R1), dynamic creation and maintenance of non-overlapping regions (R2), and internal loops of regional situation recognition and action (R3). Figure 6.1 summarizes these ideas.

6.3 Dynamic Decentralization Domains in Practice

From the previous discussion, we further refine the requirements and extrapolate the design elements of an Application Program Interface (API) supporting the decentralized computation we need:

- concerning CCTs (cf. R1)
  - we use a CCT to model a collective sensing task partitioned into multiple *sensing domains* (i.e. DDs), where each sensing domain has a *centre* and an *extension* in space;
  - both the extension in space and the centre can change dynamically to improve the way the underlying phenomenon is being tracked, through selection of an appropriate leading node, definition of a metric (which can be other than the spatial distance), and definition of a granularity.
- concerning partitioning into DDs (cf. R2) and activity within a DD (cf. R3)
  - sensing domains for a single measure must not overlap, to avoid duplicate sampling and undesired interference (overlapping can be achieved through multiple CCTs, or by using a mixed custom metric);
  - inside a single sensing domain, a strategy is defined to collect the sensor readings;
  - the decentralized sensing will output the collectively-sensed result and the identifier of the device closer to the area centre;
  - the set of actions/actuations to perform may vary depending on the overall sensing results, could require a collective plan for coordination, and may require fine-grained information about all the results of the sensing phase.

To the best of our knowledge, no completely decentralized API/framework exists in the literature that directly satisfies the aforementioned requirements (although, of course, it can be implemented leveraging existing frameworks). Thus, we designed a Scala API, presented in Listing 6.1, which serves two roles: *(i)* to reify the sought abstractions, and hence as a specification tool for dynamic decentralization domains; and *(ii)* as a basis for a prototypical implementation on top of the ScaFi framework [Cas+21; Cas+20a], which will be presented and used in the experiments in next section. Specifically, class `DistributedSensing` denotes DDs; types `Perception`, `SituatedRecognition`,

and `Action` model sensing, reasoning, and acting operations, respectively; and `decentralisedRecognitionAndResponse` encapsulates the logic that creates multiple CCTs and manages their dynamic partitioning into DDs.

Consider the FloodWatch case study introduced in Section 6.2 as a reference scenario. We assume that several pluviometers, deployed in the city, can communicate with each other (either through cloud or directly). We want to monitor the progression of a storm hitting the city, adjusting the granularity at runtime: large areas with similar rain intensity should get clustered together; if, instead, the precipitation is spotty, each spot should form a region. In other words, we want to leverage the clustering of similarly affected areas to achieve a better global tracking of the underlying phenomenon, understand its spatial structure, and potentially exploit the information for better counteraction.

We assume that lower parts of the city are at a higher risk in case of floods. We assume that the rain gauges have a GPS sensor supporting altimetry measurement (we would like to consider this information when responding to a potential emergency). Finally, we want to consider the altimetry of an entire zone and not of a single point, and to react promptly if any rain gauge is moved to a different location: we thus use the same technique for both rain intensity and altimetry.

The application goal goes beyond sensing: when the rain in low-altitude areas is so heavy that it might cause floods, we want to:

1. propagate an alert signal to the surroundings of the area at risk, to be perceived, e.g., by smart vehicles transiting by; and
2. pre-alert the closest fire station or civil protection post to be prepared in case of actual issues.

The application logic, leveraging our API, is shown in Listing 6.2–it exemplifies the abstractions (a) and their exemplary use (b). `DistributedSensing` represents the configuration of the collective value-reading operation, that selects a leading node, expands an area of influence, and produces an area-wide result; `Action` represents a collective task enacted in response to a distributed perception; `Perception` links each distributed sensing process to the corresponding computed value (i.e., the result of the collective sensing process); `SituatedRecognition` maps collective perceptions to actual actions; `decentralizedRecognitionAndResponse` is the entry point.

6.4 Evaluation

In this section, we consider the FloodWatch case study, and show that our API can successfully be used in a challenging scenario to program a system behaviour that responds as expected to the underlying environmental phenomena.

Listing (6.1) Scala API for decentralized situation recognition

```
/* Configuration of a distributed sensing task */
class DistributedSensing[Leadership,Distance,Data](
  perceptionCenter: () => Leadership,
  localValue: () => Data,
  metric: Data => Distance,
  accumulate: UnivariateStatistics[Data],
  limit: Distance
){ def compute(): Data = /* API implementation */ }

type Perception[Data] =// Collective sensing result
Map[DistributedSensing[?, ?, Data], Data]
type Action = () => Unit // Response action type

/* Situation recognition: perception to action */
type SituatedRecognition[Data] = Perception[Data] => Set[Action]

// Actual high level API
def decentralizedRecognitionAndResponse[Data](
    sensing: Set[DistributedSensing[?, ?, Data]],
    situatedRecognition: SituatedRecognition[Data]
): Unit = { /* CCT creation and Action execution */ }
```

Listing (6.2) Example use of the API for the case study

```
// FloodWatch program
type FloodWatchSensing = DistributedSensing[ID, Double, Double]
val altimetry: FloodWatchSensing = new DistributedSensing(/**/)
val rainInnensity: FloodWatchSensing = new DistributedSensing(/**/)
val sensing = Set(altimetry, rainIntensity)

def propagateAlarm(): Action = ???
def callForHelp(): Action = ???

val response: SituatedRecognition[Double] =
  situation => {
    /* create actions related to alarms */
    val alarm: Set[Action] = ???
    /* call fire station following alarms */
    val call: Set[Action] = ???
    alarm ++ call
}
// Actual API usage
decentralizedRecognitionAndResponse(sensing,response)
```

Figure 6.2: Scala implementation of the proposed API

6.4.1 Experimental setup

We exercise the API in a challenging and realistic scenario, using open data of Toronto[2], featuring 50 water gauges samples taken in 2021. To stress-test our proposed approach with a denser network of devices, we added 300 simulated gauges, randomly positioned, whose data is interpolated from the values of the surrounding real devices. We selected the rain event that occurred on 2021-09-07, the heaviest in the available data. We used data from OpenStreetMap[3] to position 24 fire stations.

We implemented the proposed API in the ScaFi aggregate programming toolkit and simulated the scenario using the Alchemist simulator [PMV13]. In the experiment, devices compute their programs unsynchronized at a frequency of 1Hz. We define a simple metric for the actual risk of a location as the quotient of the local rain intensity on the local altitude (namely, the rainier and the lower the position, the higher the risk); we run an oracle measuring it with a fine grain across the city at each instant. As performance measure, we count how many alerts get generated and how many stations they reach. Additional gauges position and device timing drift are randomized. We ran 64 repetitions of the simulation and considered the mean results. The experiment is available and reproducible; it has been released, open-sourced[4], and permanently archived [AP22]. Figure 6.3 depicts the scenario as simulated in Alchemist.

6.4.2 Results and discussion

Figure 6.3 shows that, when conditions change, DDs adapt by changing their shape and extension to track the underlying phenomenon coherently; in response to heavy rain, close-by stations get appropriately alerted. The system macroscopically tracks the underlying phenomena: more operators get alerted when (Figure 6.4) and where (Figure 6.5) there are peaks in the signal. However, even in response to similarly high peaks the system may decide to allocate less or more resources to manage them: differences are primarily due to the system detecting different base risks (due to the altitude) or the event being strictly local.

We now give some remarks to better contextualize the contribution. Regarding applicability and generality, we observe that CCTs and partitioning into DDs enable addressing several kinds of applications in domains like computing ecosystems, wireless sensor networks (WSNs), IoT and smart city, and multi-robot/multi-agent systems—cf. the surveys in [Pia+21a; Vir+19]. Details on quantitative cost/performance considerations on this kind of paradigm, can be found in [Pia+21a;

[2] `https://bit.ly/3QciJ9i`

[3] using Overpass API `https://overpass-turbo.eu/`

[4] `https://bit.ly/3vFO9P6`

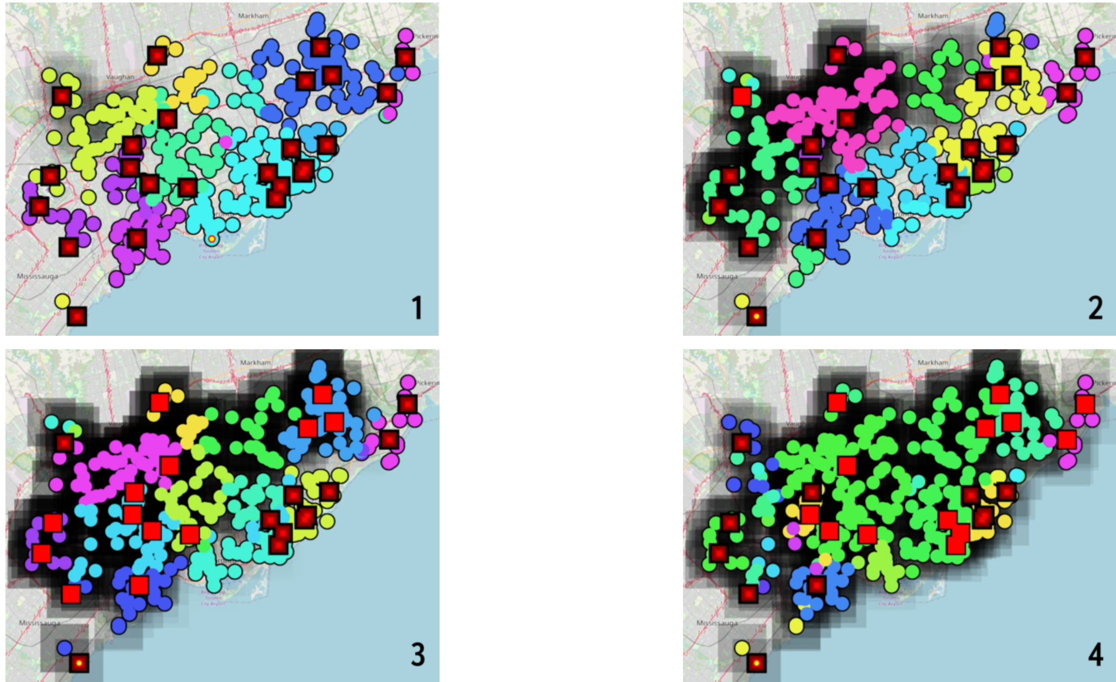


Figure 6.3: Subsequent simulation snapshots (left-to-right, top-to-bottom) of FloodWatch. Darker shadows indicate heavier rain. Black squares with a small red dot are unalerted fire stations, when at least an alert reaches them, their dot changes to a large red square. Circles represent gauges; their colours map the DDs they are subject to when measuring rainfall intensity. A video of a complete simulation run is available at `https://bit.ly/3zysMOV`.

Cas+21]: the focus of this article is on programming abstractions for CPSW following a language-based software engineering approach [Gup15].

6.5 Final Remarks

Mechanisms based on decentralization and self-organization are intensely researched and expected to play crucial roles in next-generation applications involving CPSW. In this work, to turn decentralized activities into actionable notions, we propose a high-level programming model for situation recognition and action that originally integrates recent developments in collective adaptive computing. The idea is to expose a declarative API featuring *(i)* concurrent collective tasks which overlap in space and *(ii)* non-overlapping decentralization domains with inner information flows-based feedback loops. We implement the API in Scala by mapping CCTs and DDs to ScaFi aggregate computations, then show the approach's effec-

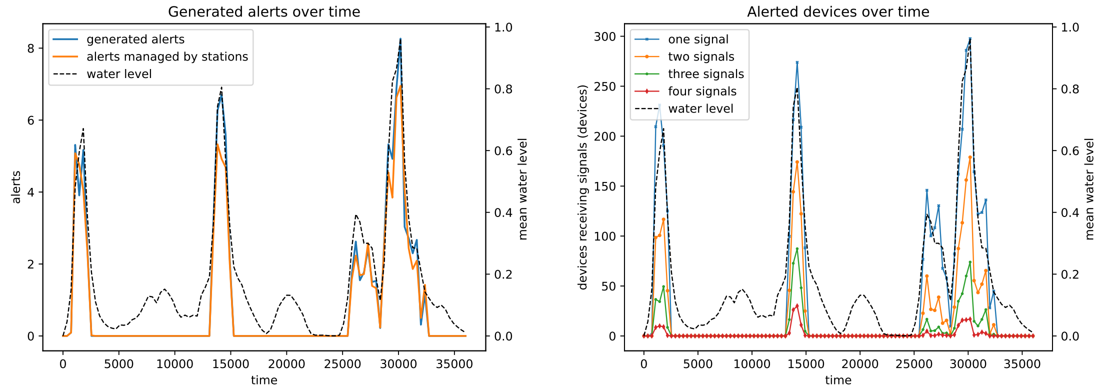


Figure 6.4: Simulation results, showing, with time, the average rainfall intensity (dotted black line), the number of operator stations receiving alerts (left) and the breakdown by number of alerts received per station (right).

tiveness through a case study in flood monitoring and control. Results show that programs expressed declaratively through the API yield DDs that can adapt to properly handle distributed monitoring and action.

This work focused on designing and programming decentralized systems. We believe that this level of control is instrumental for properly structuring collective adaptive behaviour to steer desired emergents. Building upon these foundational abstractions, it becomes feasible to develop more intricate, high-level, and domain-specific constructs. Towards this direction, in the following chapter, we introduce MacroSwarm, a macro-programming API tailored for swarm-like system steering and control.

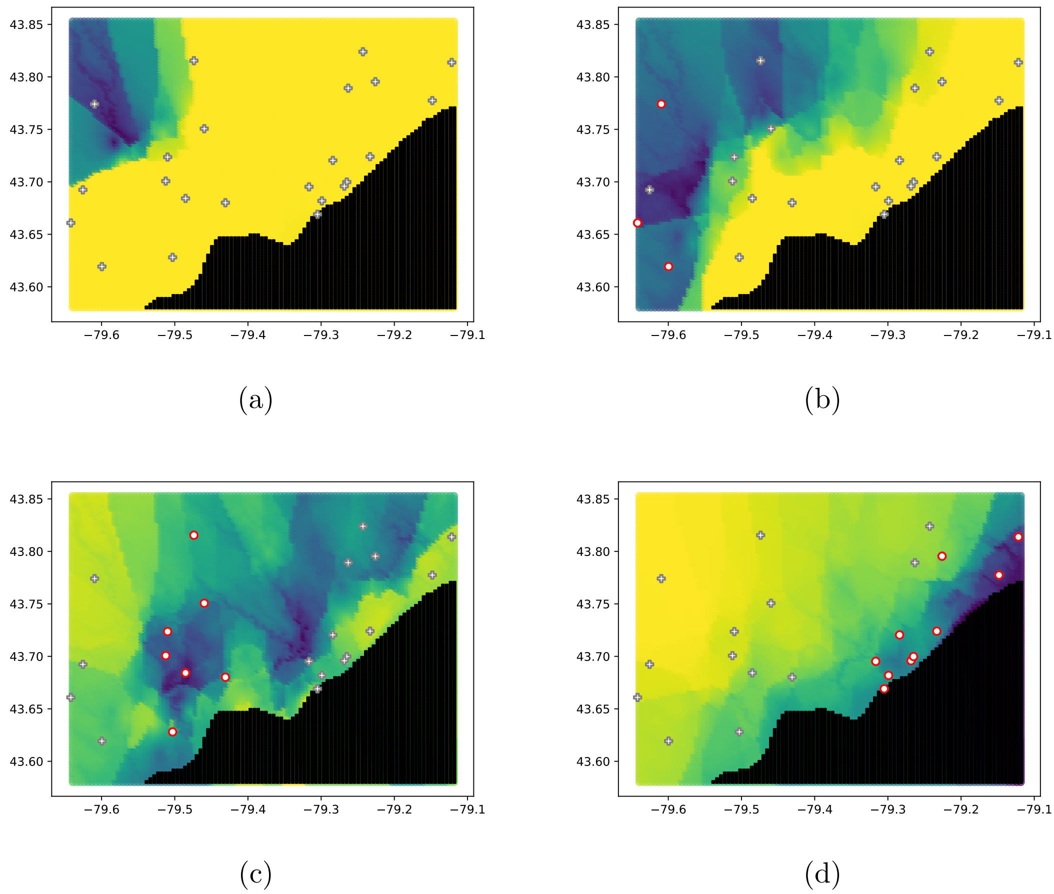


Figure 6.5: Ability to track the risk spatially. Charts show risk (the darker, the higher) as estimated in real-time by an oracle using altitude and rainfall intensity. Stations are depicted with a red circle, and those alerted are filled in white. Solid black areas are non-land. Alerted stations are indeed those closest to the zones of the highest risk.

Chapter 7

Patterns: Coordinated Movements and Decision Making

What are the patterns that can be used to express coordinated movements and decision-making in a swarm?
Can we design a language-based approach to express these patterns in a formal-yet-practical way?
How can we evaluate the effectiveness of such an approach?
–**RQ4**

Contents

7.1 Motivation 135
7.2 API Design 136
7.2.1 Movement blocks 136
7.2.2 Flocking blocks 138
7.2.3 Leader-based blocks 139
7.2.4 Team formation blocks 139
7.2.5 Pattern formation blocks 140
7.2.6 Swarm Planning blocks 141
7.3 Evaluation 143
7.3.1 Case Study: Find and Rescue 143
7.3.2 Discussion 146
7.4 Related Work 148

7.5 **Final Remarks** **149**

In CPSW domains, a prominent research problem is how to effectively engineer *swarm behaviour* [Bra+13], i.e., how to promote the emergence of desired global-level outcomes with inherent robustness and resiliency to changes and faults in the swarm or the environment. Complex patterns can emerge through the interaction of simple agents [Bon+99] and centralized approaches can suffer from scalability and dependability issues: as such, we seek for an approach based on suitable distributed coordination models and languages to steer the micro-level activity of a possibly large set of agents.

Though several approaches and languages have been proposed for specifying or programming swarm behaviour [Ash+07; CNS21; DK18; Kos+20; KL16; Lim+18; Mot+14; PB16b; Yi+20], a key feature that is generally missing or provided only to a limited extent is *compositionality*, namely, the ability to combine blocks of simple swarm behaviour to construct swarm systems of increasing complexity in a controlled/engineered way. Additionally, most existing approaches tend to be pragmatic, not formally founded and quite ad-hoc: they enable the construction of certain types of swarm applications but with limited support for analysis and principled design of complex applications (e.g. [Lim+18; DK18; PB16b; CNS21]). Exceptions that provide a formal approach exist, but they are typically overly abstract, requiring additional effort to code and execute swarm control programs [Luc+19].

In this chapter, we introduce a formally-grounded API, expressive and practical enough to concisely and elegantly encode a wide array of swarm behaviours incorporating both coordinated movement and collective decision-making. In the eye of the language-based approach, we based this design on aggregate computing: each block of swarm behaviour is captured by a purely functional transformation of sensing fields into actuation fields including movement vectors, and such a transformation declaratively captures the state/computation/interaction mechanisms necessary to achieve that behaviour. Practically, such specifications can be programmed as Scala scripts in the ScaFi framework [Cas+22a], a reference implementation for field-based coordination and aggregate computing. Accordingly, we present MacroSwarm, a ScaFi-based framework to help programming with swarm behaviours by providing a set of blocks covering key swarming patterns as identified in literature [Bra+13]: flocking, leader-follower behaviours, morphogenesis, and team formation. To evaluate MacroSwarm, we show a use case that leverage our API in a simulated environment based on the Alchemist multi-agent system simulator.

7.1 Motivation

Engineering the collective behaviour of swarms is a significant research challenge [Bra+13].

A main distinction in engineering collective behaviour is between *centralized* (*orchestration-based*) and *decentralized* (*choreographical*) approaches. In the former category, programs generally specify tasks and relationships between tasks, and these descriptions are used by a centralized entity to command the behaviour of the individual entities of the swarm. By contrast, decentralized approaches do not rely on any centralized entity: each agent is driven by a control program and the resulting execution is decentralized (e.g., based on interaction with neighbours, like in Meld [Ash+07]). In this chapter, we focus on *decentralized* solutions, since they support resilience and scalability by avoiding single points of failure and bottlenecks.

In the general context of behaviour-based swarm design, researchers have pointed out various issues [Bra+13; DTT20] like a general lack of *top-down* design methods of collective behaviours (cf. the scientific issue of "emergence programming" [Var+15] and "self-organization steering" [Ger+20]), the problem of formal verification and validation [Luc+19], heterogeneity, and operational/maintenance issues (e.g., scalability, adaptation, and security).

Specific challenges can be found in the context of specific kinds of systems, such as (micro) aerial swarms [Abd+21; Cop+20], specific domains, like agriculture [Alb+22], or specific kinds of tasks, like simultaneous localisation and mapping (SLAM) [KGB21].

To address top-down swarm programming, an approach should provide the means to define and compose blocks of high-level swarm behaviours. Regarding the kinds of blocks that can be provided, it is helpful to look at proposed taxonomies of collective/swarm behaviour. In a prominent survey on swarm engineering [Bra+13], collective behaviours are classified into;

- spatially-organising behaviours (e.g., pattern formation, morphogenesis);
- navigation behaviours (e.g., collective exploration, transport, and coordinated motion);
- collective decision-making (e.g., consensus achievement and task allocation);
- others (e.g., human-swarm interaction and group size regulation).

Finally, we observe in the literature a rather sharp distinction between approaches leveraging formal methods for specifying swarm behaviour [Luc+19], also enabling verification, and more pragmatic approaches offering concrete Domain Specific Languages (DSLs) that are more usable. In a recent survey on formal specification

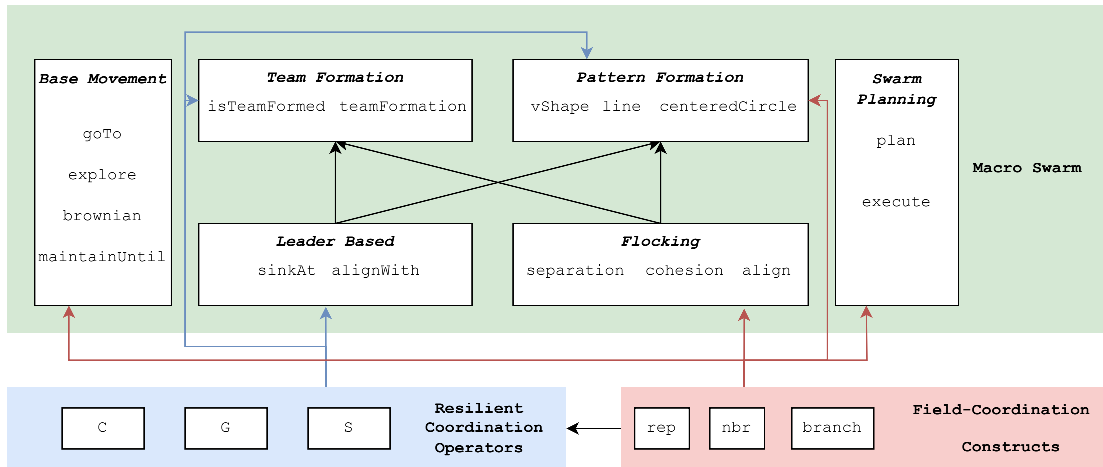


Figure 7.1: MacroSwarm: architecture overview. The black boxes contained in the green rectangle represent the main modules of the library.

methods for swarm robotics [Luc+19] it is reported that a major limitation lies in (i) the tooling and (ii) the formalization of the "last step" of passing from a formal model to program code. Hence, we seek here an approach that combines the benefits of formal methods and the pragmatism of concrete DSLs.

In summary, this work is motivated by the need for an approach for *formal-yet-practical top-down behaviour-based design of decentralized swarm behaviour*.

7.2 API Design

This section presents the MacroSwarm approach and API. In particular, we describe its overall architecture and the main blocks exposed by the API (summarized in Figure 7.1), which support the specification of a wide range of high-level swarm behaviours. The key idea in the design of MacroSwarm lies in the representation of a swarm behavioural unit as a function mapping sensing and parameter fields to actuation fields (often, velocity vectors). We have organized the API into multiple *modules*, capturing logically related sets of behaviours, and comprising more fundamental and reusable sets of behaviours as well as more application-specific sets (e.g., related to movement or team formation).

7.2.1 Movement blocks

These blocks control the movement of individual agents within the swarm. The simplest movement expressible with MacroSwarm is a collective constant movement (Figure 7.2a), described through a tuple like `Vector(x,y,z)` that devises

the velocity vector of the swarm:

```
Vector(2.5, 0, 0) // a constant field which is the same for all the agents
```

This vector must then be appropriately mapped the right electrical stimulus for the underlying engine platform of the mobile agent of interest. On top of that, this module exposes several blocks to explore an environment. Particularly, the `brownian` block produces a random velocity vector for each evaluation of the program. In addition to that simple logic, there are movements based on GPS like `goTo` (produces a velocity vector that eventually moves the system to sink at one single point) and `explore` (produces a velocity vector that lets the system explore a rectangle defined through `minBound` and `maxBound`). The last one is based on temporal blocks, like `maintainTrajectory` and `maintainUntil`. The former allows the systems to maintain a certain velocity for the time specified. At that moment, a new velocity is generated according to the given strategy. The latter, instead, is used to maintain a certain velocity until a condition is met (e.g., a target position is reached). This module also exposes an `obstacleAvoidance` block (Figure 7.2d), which creates a vector pointing away from obstacles.

Even if these blocks are quite simple, it is still possible to combine them to create interesting behaviours. For instance, the program

```
(maintainVelocity(browian()) + obstacleAvoidance(sense("obs"))).normalize
```

expresses a collective behaviour in which the nodes will explore the environment, while avoiding any obstacles perceived through a sensor. Notice how the composition is achieved by simply summing the computational fields produced by the sub-blocks. The expression `v.normalize` yields `v` as a unit vector (of length 1), while keeping the same direction—useful when combining several vectors together. A summary of the blocks exposed by this module is reported in the following listing:

```
// Movement library
def brownian(scale: Double): Vector
// GPS Based
def goTo(target: Point3D): Vector
def explore(minBound: Point3D, maxBound: Point3D): Vector
// Temporal Based
def maintainTrajectory(trajectory: => Vector)(time: FiniteDuration):Vector
def maintainUntil(direction: Vector)(condition: Boolean): Vector
// Obstacle Avoidance
def obstacleAvoidance(obstacles: List[Vector]): Vector
```

7.2.2 Flocking blocks

In a CPSW, it is often necessary to coordinate the movement of the entire swarm, rather than just individual agents, to achieve emergent behaviours, and ensure that the nodes move *cohesively*, avoid *collisions*, and strive to be *aligned* in a common direction. Therefore, in this module, we have implemented the main blocks to support the *flocking* of agents. Several models are available in the literature for this purpose. Particularly, MacroSwarm exposes the Vicsek [Vic+95], Cucker-Smale [CS07], and Reynolds (Figure 7.2e) [Rey87] models. We have also exposed the individual blocks to implement Reynolds, which are `cohesion`, `separation`, and `alignment`. These blocks can be used individually by higher-level blocks to implement specific behaviours (e.g., following a leader while avoiding collisions).

Another essential aspect that emerges at this level is the concept of a *variable neighbourhood*. Indeed, it may happen that the logical neighbourhood model used by aggregate computing does not match the one used to coordinate the agents. Thus, the node's visibility can be more *restrictive* or *extensive* according to the neighbourhood model applied. In particular, in the case of Reynolds, it is typical for the separation range to be different from that of alignment. Therefore, the flocking blocks accept a "query" strategy towards a variable neighbourhood. The main implementation of these queries are:

- `OneHopNeighborhood`: the same as the aggregate computing model;
- `OneHopNeighborhoodWithinRange(radius: Double)`: it takes all the nodes in the neighbourhood within the given range.
- `LongRangeNeighborhood(radius: Double)`: it expands the range of aggregate computing communication by spawning an aggregate process for each node that expands itself within the range passed.

The flocking models are typically described by an iterated function in which the velocity at time $t+1$ depends on the velocity at time t. Taking as an example the Vicsek rule, it is described as: $v_i(t+1) = \frac{\sum_{j \in \mathcal{N}} v_j(t)}{|\mathcal{N}|} + \eta_i(t)$ where $\mathcal{N}$ is the neighbourhood of the node i at time t, $v_i(t)$ is the velocity of the node i at time t, and $\eta_i(t)$ is a random vector that models the noise of the model. For this reason, each block receives the previous velocity field as a parameter, rather than encoding it internally within each block. This is because the previous velocities may be influenced by other factors, such as constant movements or a target position. Typical usage of this operator follows the following schema:

```
rep(initialVelocity) { oldVelocity => flockingOperator(oldVelocity, ..) }
```

For example, the following program describes a collective movement in which the nodes try to reach the position `(x,y)` while maintaining a distance of `k` meters from one another:

```
rep(Point2D.Zero) {
  v => (goTo(Point2D(x, y)) +
       separation(v, OneHopNeighbourhoodWithinRange(k))).normalize
}
```

7.2.3 Leader-based blocks

These blocks allow agents to follow a designated leader. The idea behind leadership in swarm systems is that a leader can act as a coordinator, influencing the followers that recognize it as such. In the context of aggregate computing, leaders are typically defined as Boolean fields holding `true` for leaders and `false` for non-leaders. Leaders can be predetermined (i.e., nodes with certain characteristics), virtual (i.e., nodes that do not exist in the system but are simulated for collective movement steering), or chosen in space (e.g., using the `S` block—see Section 3.3.2). A leader can be thought of as creating an *area of influence*, affecting the actions of its followers. Currently, we have identified `alignWithLeader` and `sinkAt` (Figure 7.2b) as essential blocks. The former propagates the leader's velocity throughout its area of influence (e.g., via `G`—see Section 3.3.2), with followers adjusting their velocity to it. However, sometimes it may also be desirable to create a sort of attraction towards the leader, so that the nodes remain cohesive with it. For this reason, the `sinkAt` block creates a computational field in which nodes tend to move towards the leader. These blocks are useful for higher-level blocks, such as those associated with the creation of teams or spatial formations.

7.2.4 Team formation blocks

These blocks allow agents to form *teams* or sub-groups within the swarm, useful e.g. for work division or situations requiring intervention by a few agents. In general, the formation of a team creates a "split" in the swarm logic, conceptually creating multiple swarms with potentially different goals (cf. Figure 7.2c). One way to create teams is by using the `branch` construct (see Section 3.3.2). For example, the following program,

```
def alignVelocity(id: Int) =
  alighWithLeader(id == mid(), rep(browian())(x => x)
branch(mid() < 50) { alignVelocity(0) } { alignVelocity(50) }
```

creates two groups, each of which follows a certain velocity dictated by the leaders (0 and 50).

Other times, one needs to create teams based on the spatial structure of the network or when certain conditions are met. The `teamFormation` block supports this scenario. By internally using `S`, it allows for the creation of teams based on certain spatial constraints expressed through parameters `intraDistance` (i.e., the distance between team members) and `targetExtraDistance` (i.e., the size of the leader's area of influence). It is also possible to create teams based on predetermined leaders, denoted explicitly by Boolean fields. Moreover, since team formation may take time to complete, or require conditions to be met (e.g., that at least N members are present, or that the minimum distance between all nodes is less than a certain threshold), we also parameterize `teamFormation` by a `condition` predicate. An example of built-in predicate is `isTeamFormed`, which verifies that each node under the influence of the leader has a `necessary` a number of neighbours within a `targetDistance` radius. An example is as follows.

```
teamFormation(targetIntraDistance = 30, // separation
  targetExtraDistance = 300, // influence of the leader
  condition = leader => isTeamFormed(leader, targetDistance = 40)
).velocity // use the velocity vector to create the Team
```

Each team must refer to a single leader, who can coordinate the associated nodes (using the APIs exposed by the **Leader Based Block**). In particular, to execute a certain behaviour within a team, the `insideTeam` method must be used. Given the ID of the leader to which a node belongs, this method can define the movement logic relative to that leader. For instance, this code aligns the followers with a velocity generated by a leader,

```
team.insideTeam{ leader =>
  alignWithLeader(leader) {
    rep(brownian())(x => x))
  }
}
```

7.2.5 Pattern formation blocks

Team formation blocks can be used to create groups of agents with certain characteristics. However, sometimes we are also interested in the *spatial structure* of the group. In swarm behaviours, the spatial structures of the teams can be instrumental in performing certain tasks (e.g., coverage or transportation tasks). In MacroSwarm some of the most idiomatic spatial structures are available.

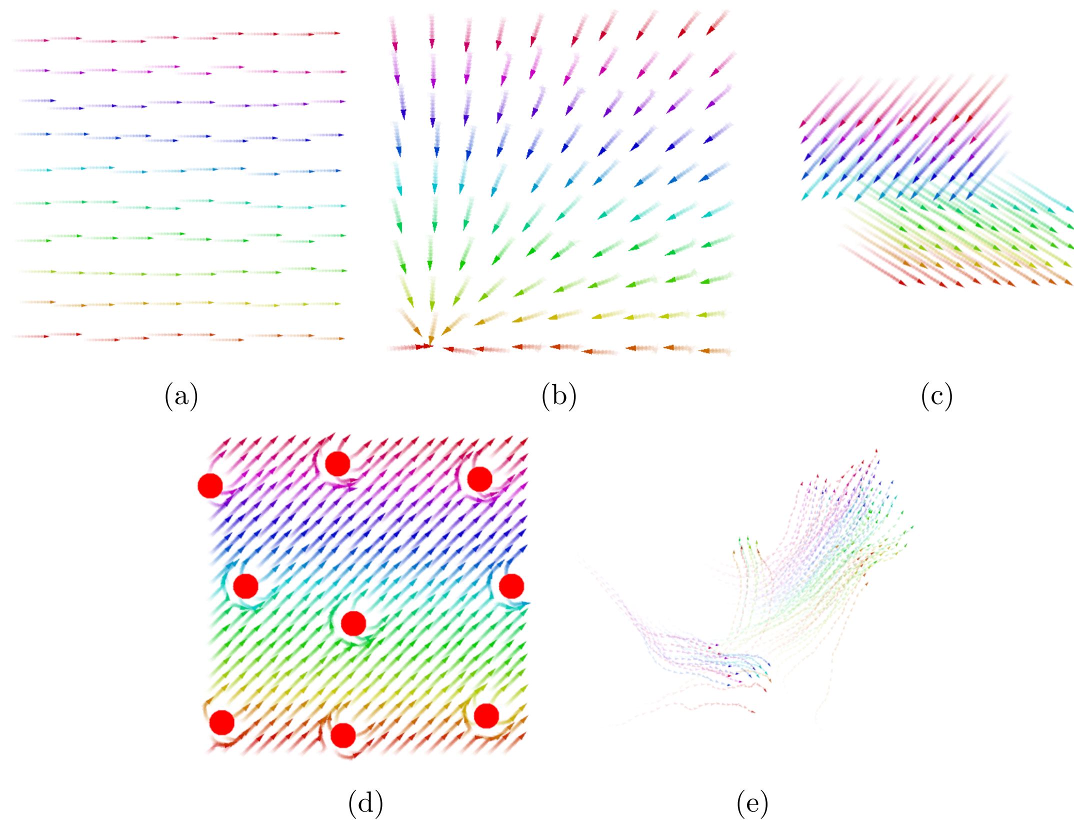


Figure 7.2: Overview of swarm behaviours expressible with MacroSwarm.

The implementation is as follows. First, the formation of structures is based on the presence of a leader that collects the hop-by-hop distances of their followers (leveraging `G` and `C`) and sends them a direction in which they should go to form the required structure (using `G`).

The structures currently supported (Figure 7.3) are v-like shapes (`vShape`), lines (`line`), and circular formations (`centeredCircle`). These structures are *self-healing*: if there is a disturbance of the structure, the group tends to reconstruct itself and return to a stable structure. Additionally, it is assumed that the leader has his own speed logic. In this way, the group will follow the leader maintaining the chosen structure.

7.2.6 Swarm Planning blocks

With the previous blocks available, there is a need for a handy mechanism to express a series of *plans* that change over time and move the swarm towards different targets. For this reason, MacroSwarm also exposes the concept of *swarm plan-*

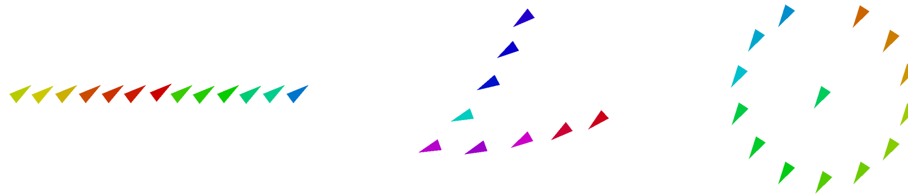


Figure 7.3: Examples of the supported patterns. From left to right: line formation, v-like formation, and circular formation.

ning. The idea is to express a series of plans (or missions) defined by a *behaviour* (i.e., the logic of production of a velocity vector) and a *goal* (defined as a boolean predicate condition). At any given time, the swarm will be executing a certain sub-plan, which will be considered complete only when the boolean condition is satisfied. At this point, the swarm will follow the next objective described by the overall plan. The exposed API allows for the creation of these collective plans in the following way:

```
execute.once {
  plan(goTo(goalOne).endWhen(isClose(goalOne)),
  plan(goTo(goalTwo).endWhen(isClose(goalTwo)),
}.run() // will trigger the execution of the plan
```

This snippet creates a plan in which the nodes will first go to `goalOne`, and once reached (`isClose` verifies that the node is close enough to the point passed), it will move on to the next objective `goalTwo`. Since it is specified that the mission is executed `once`, after the completion of the last plan, the group will stop moving. To make the group repeat the plan, the `repeat` method can be used instead of `once`. Note that there is no coordination between agents in the above code, but you can enforce it using lower-level blocks (e.g., flocking or team-based behaviours). For example, MacroSwarm enables describing a swarm behaviour where:

1. a group of nodes gathers around a leader,
2. the leader brings the entire group towards the `goalOne`,
3. the leader brings the entire group towards the `goalTwo`.

This can be described using the following code:

```
execute.once( // if it is repeated, you can use `repeat'
  plan{sinkAt(leaderX)}.endWhen{
     isTeamFormed(leaderX, targetDistance=100)
  },
```

```
  plan(goTo(goalOne)).endWhen{ G(leaderX, isClose(goalOne), x => x)},
  plan(goTo(goalTwo)).endWhen{ G(leaderX, isClose(goalTwo), x => x)},
).run()
```

The use of `G` in this way is a recurrent pattern, and in ScaFi it is exposed through the `broadcast[T](center: Boolean, value: T): T` block.

7.3 Evaluation

To validate the proposed approach and API we define a simulated *find-and-rescue* case study, to show the ability of MacroSwarm to express complex swarm behaviours (Section 7.3.1). Then, we discuss the results of the case study and the applicability of the proposed approach in real-world scenarios (Section 7.3.2).

7.3.1 Case Study: Find and Rescue

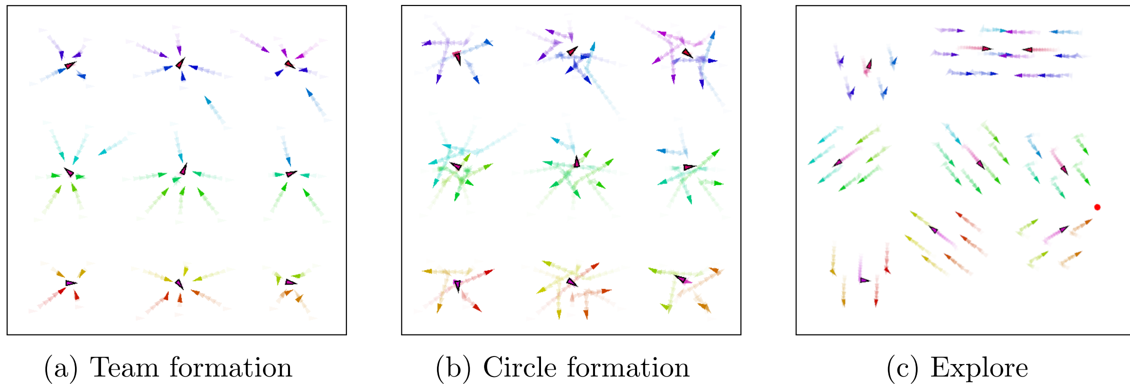
In our scenario, we want a fleet of drones to patrol a spatial area. In the area, dangerous situations may arise (e.g., a fire breaks out, a person gets injured, etc.). In response to these, a drone designated as a *healer* must approach and resolve them. Exploration must be carried out in groups composed of *at least* one healer and several *explorers*, who will help the healer identify alarm situations.

7.3.1.1 Goal

The goal of the proposed case study is to demonstrate the effectiveness of the proposed API in terms of *expressiveness* (i.e., the ability to describe complex behaviours easily) and *correctness* (i.e., the described behaviour collectively does what is expressed). For the first point, since it is a qualitative metric, we will show the development process that led to the implementation of the produced code, demonstrating its ease of understanding. For the second point, since deploying a swarm of drones is costly, we will make use of simulations to verify that the program is functioning correctly both qualitatively (e.g., observing the graphical simulation) and quantitatively (i.e., extracting the necessary data and computing metrics that allow us to understand if the system behaves as it should).

7.3.1.2 Setup

Initially, 50 explorers and 5 healers are randomly positioned in an area of $1km^2$. Each drone has a maximum speed of approximately 20 km/h and a communication range of 100 meters. The alarm situations are randomly generated at different times within the spatial area in a $[0, 50]$ minutes time-frame. Each simulation run



(a) Team formation (b) Circle formation (c) Explore

Figure 7.4: The first phases of the scenario described in Section 7.3. At the beginning, the system is split into teams; afterwards, the teams assume a spatial formation (circular, in this case); finally, the teams start exploring the overall area.

lasts 90 minutes, during which we expect the number of alarm situations to reach a minimum value. The node should form teams of at least one healer and several explorers, maintaining a distance of at least 50 meters between the node and the leader

7.3.1.3 Implementation details

To structure the desired swarm behaviour, we break the problem into parts:

1. the swarm must split into teams regulated by a healer, who works as a *leader* (Figure 7.4a);
2. teams must assume a spatial formation promoting the efficiency of the exploration (Figure 7.4b);
3. the teams must explore the overall area (Figure 7.4c);
4. when any node detects an alarm zone, it must point that to the healer;
5. the healer node approaches the dangerous situation to fix it;
6. then, the team should return to the exploration phase.

We now describe the implementation of each part, leveraging the MacroSwarm API. First, for creating teams, we can use the **Team Formation** blocks:

```
val teamFormedLogic =
  (leader: ID) => isTeamFormed(leader, minimumDistance + confidence)
def createTeam() =
  teamFormation(sense("healer"), minimumDistance, teamFormedLogic)
```

where minimumDistance is the minimum distance between nodes during the team formation phases and confidence is the confidence interval used to check if the team is formed through the isTeamFormed method. Each team then should follow the aforementioned steps, expressible using the **Swarm Planning** API:

```
def insideTeamPlanning(team: Team): Vector =
 team.insideTeam {
  healerId =>
   val leading = healerId == mid() // team leader
   execute.repeat(
    plan(formation(leading)).endWhen(circleIsFormed), // shape formation
    plan(wanderInFormation(leading)).endWhen(dangerFound), // exploration
    plan(goToHealInFormation(leading, inDanger)).endWhen(dangerReached),
    // healing
    plan(heal(healerId, inDanger)).endWhen(healed(dangerFound))
   ).run() // repeat the plan
 }
```

The first step is the formation of the teams, based on method formation which internally uses centeredCircle to place the nodes in a circle around the leader node. The function circleIsFormed verifies whether the nodes are in a circle formation, i.e., that the distance between any node and the leader is less than radius (set to 50 meters in this scenario). The second step is the exploration phase, implemented by method wanderInFormation, which uses the explore function to move the nodes to a random direction within given bounds while keeping the circle formation. This leverages centeredCircle, passing the movement logic of the healer (leader) to the block. Exploration will go on until someone finds a danger node, denoted by predicate dangerFound. This internally uses C and G to collect the danger nodes' positions and share them within the team:

```
def dangerFound(healer: Boolean): Boolean = {
  val dangerNodes =
    C(sense("healer"), combinePosition,
    List(sense("danger")), List.empty)
  broadcast(healer, dangerNodes.nonEmpty)
}
```

The third step is the movement towards the danger node, which is implemented by the goToHealInFormation method, which uses again the centeredCircle function with a delta vector that moves the leader node towards the danger node. inDanger is computed similarly to dangerFound, but, in this case, the position will be shared instead. dangerReached is a Boolean field indicating if the healer node is close enough to the danger node. The last step is the healing of the danger node, which is modelled as an actuation of the healer. The rescue ends when the

danger node is healed. As a final note, we also want the nodes to be able to avoid each other when they are too close, even if they are not in the same team. For this, we leverage the **Flocking** API the `separation` block outside the team logic. Then, the main program is as follows:

```
val team = createTeam()
rep(Vector.Zero) { v =>
  insideTeamPlanning(team) +
  separation(v, OneHopNeighbourhoodWithinRange(avoidDistance))
}.normalize
```

This program shows that the API is flexible enough to create complex behaviours handling various coordination aspects.

7.3.1.4 Results

We validated the results by effectively running Alchemist simulations, publicly available at `https://zenodo.org/badge/latestdoi/611692727`. We launched 64 simulation runs with different random seeds: Figure 7.5 shows the average results obtained. Furthermore, we extracted the following data:

- *intra-team distance*: after an initial adjustment phase, the system should converge to an average distance of 50 meters (Figure 7.5a);
- *minimum distance between each node*: as we want to avoid collisions, the minimum distance between two nodes should always be greater than zero (Figure 7.5b);
- *number of nodes in danger*: we expect the nodes in danger to increase up to 50 minutes and then decrease, tending towards zero (Figure 7.5c).

The results (Figure 7.5) show that the system can achieve the expected outcomes.

7.3.2 Discussion

Despite its simplicity, this use case allowed us to demonstrate the capability of MacroSwarm, both in qualitative terms (i.e., the produced code is simple and understandable) and quantitative terms (i.e., the data show that the swarm follows the given instructions correctly).

That being said, there are several things to consider when using the library in real-world contexts. Ours is a top-down approach, in which we have defined an evaluation and implementation system that is general enough to be executed in various multi-agent systems. Specifically, we require that at least:

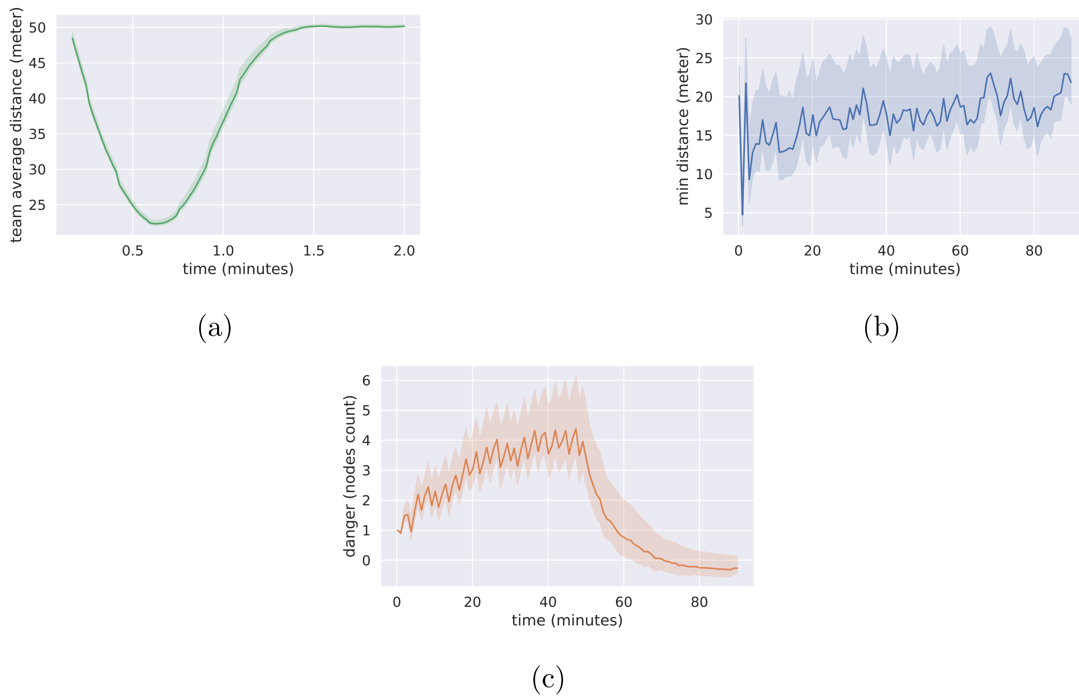


Figure 7.5: Quantitative plots of the simulated scenario. Figure 7.5a shows the average team distance in the first two minutes. Figure 7.5b shows the minimum distance between nodes. Figure 7.5c shows the nodes in danger through time. Since we ran several simulations, the lines show the average values, whereas the area around the lines shows the confidence interval throughout the simulations.

i) nodes can perceive and interact with neighbours and approximate a direction vector to each of them;

ii) they can move in a specific direction with a certain velocity; and

iii) they can perceive distance and direction for certain obstacles.

As for point *i)*, this can be developed using specific local sensors (e.g., range and bearing systems [Bil+22]), by using GPS, by approximating distances using cameras mounted on each drone, or by using Bluetooth direction finding [SW22]. Concerning the point *ii)* the velocity vector can be mapped to the motors of the UAVs, or the motor's wheels of the ground agents [KB91], so it can be easily implemented in real-case scenarios. Finally, concerning *iii)*, there are several solutions for perceiving the direction of obstacles by leveraging various sensors, like *Laser Imaging Detection And Ranging (LIDAR)* systems [Pen+15].

That being said, we know that the reality gap for real-world scenarios could introduce divergences from the behaviours shown, as the used simulator, although

general, does not simulate many aspects of reality, such as communication delay, friction, and possible perception errors. We aim to test the API in more realistic simulators (like Gazebo [KH04]) or real systems as a future work.

7.4 Related Work

Related programming approaches for swarms include Meld [Ash+07], Buzz [PB16b], Voltron [Mot+14], TeCoLa [KL16], Dolphin [Lim+18], MapleSwarm [Kos+20], PARoS [DK18], Resh [CNS21], and [Yi+20]. In the following, we review the works that are more related to MacroSwarm, which are those for expressing *decentralized* behaviours.

Buzz [PB16b] is a mixed imperative-functional language for programming swarms. In Buzz, swarms are first-class abstractions: they can be explicitly created, manipulated, joined (e.g., based on local conditions), and used as a way to address individual members (e.g., for tasking them). For individual agents, the language provides access to local features and the local set of neighbours, for interaction. For swarm-wide consensus, a notion of *virtual stigmergy* is leveraged, based on distributed tuple spaces. Buzz is designed to be an extensible language since new primitives can be added. Indeed, Buzz is based on a set of quite effective but ad-hoc mechanisms. By contrast, MacroSwarm uses few general and expressive primitives, and supports swarm programming through a library of reusable, composable blocks. Additionally, MacroSwarm can leverage theoretical results from field calculi [Vir+19; Vir+18], making programs amenable for formal analysis.

Voltron [Mot+14] is a programming model for team-level design of drone systems. It represents a group of individual drones through a *team abstraction*, which is responsible for the overall task. The details of individual drone actions and their timing are delegated to the platform system during runtime. The programmer issues *action commands* to the drone team, along with *spatio-temporal constraints*. The tasks in Voltron are associated with spatial locations, and the team self-organises to populate *multisets of future values* that represent the task's eventual result at a specific location. However, Voltron is imperative in nature, limiting the compositionality of team-level behaviours.

Meld [Ash+07] is a logic-based language for programming modular ensembles, for systems where communication is limited to immediate neighbours. It leverages *facts with side-effects* to handle actuation, *production rules* to generate new facts from existing facts, and *aggregate rules* to combine multiple facts into one fact by folding (e.g., maximization or summation). The runtime deals with the communication of facts and the removal of invalidated facts. The declarativity and logical foundation make Meld an interesting macroprogramming system; however, it is not clear how it can scale with the complexity of general swarm behaviour.

Indeed, it is mainly adopted for shape formation and self-reconfiguring ensembles.

Finally, we mention another category of related works, which are *task orchestration languages* for swarms (e.g., TeCoLa [KL16], Dolphin [Lim+18], MapleSwarm [Kos+20], PARoS [DK18], Resh [CNS21], and [Yi+20]): they adopt quite a different approach that leverages centralized entities to control the activity of the swarm members based on the provided task descriptions.

7.5 Final Remarks

This chapter presents a comprehensive framework for top-down swarm programming, offering modular building blocks that encapsulate prevalent decentralized swarm behaviours.

We delineate the key contributions of this work as follows:

1. A programming model founded on the concepts of *collective behaviours* and *swarm capabilities*;
2. A library of modular, reusable building blocks designed for swarm programming and grounded in field calculus;
3. A case study that validates the expressiveness and efficacy of the framework herein proposed.

The discussions in Chapters 5 to 7 contribute to the broader vision of developing a comprehensive set of design pattern for CPSW. This aims to bridge the gap between domain-specific problems and their concrete implementations.

In the subsequent chapter, we introduce a novel programming model for CPSW, which addresses temporal considerations through a functional reactive programming paradigm.

Chapter 8

Language: Reactive-based collective computations

How can we design a reactive programming model for aggregate computing?
Why is it important to decouple the logic from the scheduling of the program?
How can we experimentally evaluate the benefits of reactivity and resource usage?
– **RQ1, RQ4**

Contents

8.1 Motivation 152
8.1.1 Self-organization Engineering Approaches 152
8.1.2 Functional Reactive Programming 153
8.2 FRASP Programming Model 156
8.2.1 System Model and (Reactive) Execution Model 156
8.2.2 Programming Abstractions and Primitives 157
8.2.3 Paradigmatic Examples: Self-Healing Gradient & Channel 159
8.3 Implementation 162
8.3.1 Goals 162
8.3.2 Architecture 164
8.3.3 Implementation details 164
8.4 Evaluation 165
8.4.1 Goals 165

8.4.2 Experimental Setup 167
8.4.3 Results and Discussion 169
8.5 Final Remarks 171

Building *artificial self-organising systems* exhibiting *collective intelligence* is a relevant research challenge spanning science and engineering [PB15; Ger07; SSP13; NJW20]. A central problem lies in *driving the (emergence of) self-organising behaviour of a collection of agents or devices*—a goal also referred to through terms like "guided self-organization" [Pro09], "controlled self-organization" [Sch+10], and "emergence steering" [Gia17]. This problem can be reduced to the definition of the control program that each agent has to execute [MH12].

As our language-based approach, in this chapter, we focus on an approach to self-organization, where the developer writes the self-organising control program using a suitable *macroprogramming language* [Cas23; Sen+22] (i.e., one aiming at expressing the macro-level behaviour of a system), be it general-purpose or domain-specific (e.g., explicitly tailored to robotic swarms [Bra+13] or wireless sensor networks [MP11]). Specifically, our high-level goal is to devise a programming language for self-organising CPSWs that is *expressive*, *practical*, and *declarative*—in the sense that it should allow the programmer to abstract from many operational details, to be dealt with automatically by the underlying middleware/-platform [Noo+19]. Specifically, we focus on the problem of concrete scheduling of the sub-activities of which a self-organising system can be composed. State-of-the-art languages typically leverage a *round-based* execution model, where devices repeatedly evaluate their context and control program entirely (typically in a loop or periodic, time-driven fashion). This approach is simple to reason about but limited in terms of flexibility in scheduling and management of sub-activities (and response to contextual changes). Motivated by this, and inspired by the functional reactive paradigm, in this chapter *we propose a reactive self-organization programming language that enables the decoupling of program logic from its scheduling.* In particular, as a contribution, we:

- propose a novel programming model and language, called *Functional Reactive Approach to Self-organization Programming (FRASP)*;
- provide an open-source implementation as a Scala DSL[1], leveraging the functional reactive library Sodium and inspiration from the ScaFi aggregate programming DSL [Cas+22a; Aud+23];
- experimentally evaluate the benefits of reactivity and resource usage through

[1] `https://github.com/cric96/distributed-frp`

an open-source, permanently available artefact with reproducible simulations[2].

The result is a functional reactive self-organization programming model, relying on the functional composition of behaviours, and whereby each sub-expression is amenable to independent scheduling: overall, we maintain the same expressiveness and benefits of aggregate programming while enabling significant improvements in terms of scheduling controllability, flexibility in the sensing/actuation model, and execution efficiency.

8.1 Motivation

In this section, we review approaches for self-organization engineering (Section 8.1.1), and set the goal of combining the benefits of reactive approaches with those of compositional macroprogramming models like aggregate computing. Then, we provide background on functional reactive programming (FRP) (Section 8.1.2), the paradigm we choose for our programming model, for its benefits in declarativity and automatic, configurable management of change.

8.1.1 Self-organization Engineering Approaches

Among reactive approaches is *Tuples On The Air (TOTA)* [MZ09], a programming model for decentralized peer-to-peer networks of mobile nodes or agents. It uses *tuples* to represent context information and mediate interactions between agents. In particular, tuples are *reactive*: they are associated with propagation rules that describe how tuples should be propagated to neighbours (hop-by-hop) in a network and how the content of tuples should change during propagation or in reaction to environmental events. The agents behave and coordinate through operations on tuples (e.g., insertion, read, removal, waiting) or by subscribing to tuple-related events. Other reactive approaches exist, such as the *Higher-Order Chemical Language (HOCL)*[BFR07], but they feature quite a large abstraction gap. On the other side, there are programming models based on a *round-based* execution model whereby each device repeatedly performs a complete evaluation of its control program (e.g., wrapped in a loop or scheduled in a periodic, time-driven fashion) as aggregate computing – our language of reference. Though conceptually simple, the round-based models could be more efficient, because they fully re-evaluate the context and the whole program without tracking change. Though it might be acceptable for predictable patterns of environmental change, this becomes largely suboptimal for highly variable dynamics. Indeed, the round-based

[2] https://github.com/AggregateComputing/experiment-2023-acsos-distributed-frp

approach seems to be a legacy of imperative languages or solutions featuring limited compositionality. Instead, more compositional languages like, e.g., aggregate computing languages [BPV15; BBM10; Aud+22] and Buzz [PB16b], allow building complex self-organising behaviour by composing *blocks* of simpler self-organising behaviours. Therefore, each individual block of behaviour is *potentially independent of others (i.e., independently schedulable)*, with data dependencies arising from each composition. A reactive extension of aggregate computing has been proposed in [Pia+21b], based on manually specifying dependencies and reactive policies (with configurable triggers) among different aggregate computations. Instead, a more practical approach could be decoupling programs from the specification of reactive policies and letting them only define the data dependencies between program portions. The most suitable programming approach for this is the *functional reactive programming* paradigm [Bai+13], briefly introduced in the following.

8.1.2 Functional Reactive Programming

Reactive programming [Bai+13] is a paradigm suitable for developing event-driven applications, leveraging abstractions to express (relationships between) time-varying values and automatically handle the propagation of change (cf. the paradigmatic example of spreadsheets [Bla16]). Reactive programming is often combined with the functional programming paradigm [Bai+13; Bla16] in the so-called FRP.

FRP builds on few abstractions and various combinators [WH00]. Conceptually, FRP considers *continuous time*, $Time = \{t \in \mathbb{R} \mid t \geq 0\}$. Time-varying values are called *cells* and may be conceptually modelled by generic functions of type $Cell\ a : Time \rightarrow a$. Then, *streams* are discrete-time values and may be modelled by generic functions of type $Stream\ a : [Time] \rightarrow [a]$ (where notation $[X]$ indicates sequences of Xs), namely, mapping a sequence of (increasing) sample times to a sequence of corresponding values. While cells model state, streams model state changes (or events). Then, FRP libraries provide functions (combinators) for transforming signals to signals, streams to streams, signals to streams, and streams to signals. An example of such a library is Sodium [Bla16], the Java library for FRP which we leveraged to implement FRASP as described in Section 8.3.

8.1.2.1 Background: Sodium

Sodium is primarily based on two types:

- `Cell<T>` represents a value of type `T` that changes over time;

- `Stream<T>` represents a sequence of emissions of events, each holding data of type `T`.

In addition, Sodium implements a series of *primitives* that can be used to perform transformations on cells and streams.

8.1.2.1.1 Never A stream that will never emit any events can be created by using the empty constructor of `Stream<T>`:

```
Stream<String> never = new Stream<>();
```

8.1.2.1.2 Constant A cell that will always have the given value can be created by passing a constant value to the constructor of `Cell<T>`:

```
Cell<String> helloWorld = new Cell<>("Hello World!");
```

8.1.2.1.3 Map A stream (resp. cell) can be transformed into a corresponding stream (resp. cell) through method `x.map(f)`, where `f` is the mapping function:

```
Stream<Integer> source = ...;
Stream<String> out = source.map(x -> Integer.toString(x));

Cell<Integer> source = ...;
Cell<String> out = source.map(x -> Integer.toString(x));
```

8.1.2.1.4 Merge Two streams of the same type can be merged into a single stream via the `s1.merge(s2,f)` method, where a mapping function `f` can be used to combine *simultaneous events*.

```
Stream<Integer> left = ...;
Stream<Integer> right = ...;
Stream<Integer> merged = left.merge(right, (l, r) -> l + r);
```

8.1.2.1.5 Hold A stream `s` can be converted into a cell through method `s.hold(init)`: the cell, holding `init` before the first event, will keep the value of the most recent event.

```
Stream<Integer> events = ...;
Cell<Integer> hold = events.hold(0);
```

8.1.2.1.6 Snapshot A cell can be converted into a stream through method `s.snapshot(c,f)`: the output stream will capture the values of the given cell `c` whenever the source stream `s` fires.

```
Stream<String> trigger = ...;
Cell<Integer> state = ...;
Stream<String> out = trigger.snapshot(state, (t, s) -> t + s);
```

8.1.2.1.7 Filter Method `s.filter(f)` produces a stream that emits only the events from the source stream `s` that satisfy the given predicate `f`.

```
Stream<Integer> events = ...;
Stream<Integer> out = events.filter(x -> x > 0);
```

8.1.2.1.8 Lift Two cells can be combined into one through a method `c1.lift (c2,f)`, where `f` is a combining function.

```
Cell<Integer> left = ...;
Cell<Integer> right = ...;
Cell<Integer> out = left.lift(right, (l, r) -> l + r);
```

8.1.2.1.9 Sample The sampling of a cell returns the current value wrapped by the cell.

```
Cell<String> state = ...;
String currentState = state.sample();
```

8.1.2.1.10 Switch (stream) Flattens a cell of streams into a single stream that emits whenever the active stream for the cell emits.

```
Cell<Stream<Integer>> source = ...;
Stream<Integer> out = Cell.switchS(source);
```

8.1.2.1.11 Switch (cell) Flattens a cell of cells into a single cell whose value is the value of the active cell of the wrapper cell.

```
Cell<Cell<Integer>> source = ...;
Cell<Integer> out = Cell.switchC(source);
```

Combining streams and cells yields a directed graph across which changes propagate that forms a piece of functional reactive logic. External interfacing with FRP graphs is supported by (i) *pushing events into streams and cells*, by leveraging types `StreamSink<T>` and `CellSink<T>`; and (ii) *listening to changes in streams and cells*, through method `listen(h)` attaching handler `h`.

8.2 FRASP Programming Model

This section presents the FRASP programming model from a user perspective. First, we explain the system and execution model (Section 8.2.1); then, we present the language abstractions and primitives (Section 8.2.2); finally, we show how paradigmatic examples of self-organization can be expressed in FRASP (Section 8.2.3).

8.2.1 System Model and (Reactive) Execution Model

In general, in aggregate computing, the *scheduling* of a program execution is asynchronous w.r.t. other devices and may be periodic (time-triggered) or reactive. In this work, we consider reactive scheduling and compare it with the periodic scheduling of earlier research. In reactive settings, a program may need to be re-executed any time an input changes, i.e.:

- sensor data (e.g., the temperature sensor perceives a different temperature);
- neighbour data (e.g., a device is no longer a neighbour, a message has expired, or a neighbour provides a more recent message that supersedes previous data).

A re-evaluation of the program may produce a different output and export. Furthermore, in this work, we take this reactivity scheduling of programs a step further by allowing individual *expressions* (i.e., portions of programs) to be re-evaluated when their context and inputs (e.g., dependencies on other expressions) change—thanks to the FRP approach. This idea will be shown in Example 2 and Figure 8.1.

Notice that the model is logical and may be implemented using different approaches and optimizations—e.g., a device may send a heartbeat to notify that it is still a neighbour and its data has not changed, it may send a message with only data that has changed, and so on. Also, inbound and outbound reactivity can be regulated by throttling, i.e., by accumulating a certain amount of (change in) inputs (before re-evaluating the program or parts of it) and accumulating a certain amount of (change in) outputs/exports (before executing actions and/or sending the export to neighbours).

8.2.2 Programming Abstractions and Primitives

In this thesis, we adopt the general system model devised in Part I, where a single program is used to express the collective behaviour of the entire system of devices. Specifically, the self-organising collective behaviour will *emerge* from (i) the local execution of the program by all the devices making up the system, (ii) the distributed execution (implementing the message passing), and (iii) the environment dynamics (which will affect neighbourhoods and the data perceived by sensors).

Since FRASP is implemented as a DSL internal (or embedded) in Scala, Scala types and features (e.g., functions) can be reused in FRASP programs.

8.2.2.1 Datatypes

According to the FRP paradigm, we would like to express a self-organising collective computation as a graph of reactive sub-computations. We call each sub-computation a *flow* and represent it programmatically through type `Flow[T]`[3], where `T` is the type of the output of the wrapped computation. A `Flow` is essentially a function that takes a `Context` and returns a *cell* of `Exports`, possibly depending on the exports of other `Flows`, recursively—see Section 8.3.3 for details. With abuse of terminology, we will refer to a flow as its output cell, i.e., as a time-varying value.

8.2.2.2 Local values

The simplest constructs of the language are local and atomic (i.e., that do not depend on other flows or neighbours).

- `constant(e)` returns a constant flow that always evaluates to the argument that has been passed;
- `sensor(name)` returns the flow of values produced by the sensor with the given `name`;
- `mid()`, as a shortcut to `sensor("mid")`, returns the constant flow of the device ID.

8.2.2.3 Choice

A `mux(c){t}{e}` expression returns a flow with the same output of flow `t` when the Boolean flow `c` is true and the output of flow `e` when `c` is false. E.g., code

[3] Notation: we highlight types in `brown`, primitives in `red`, derived/library constructs in `purple`, and Scala (host language) keywords in `blue`.

```
mux(sensor("temperature") > THRESHOLD)
    { constant("hot") } { constant("normal") }
```

will yield the string `"hot"` in the devices where the local temperature sensor yields a value below the given threshold and the string `"normal"` otherwise.

8.2.2.4 Interaction with neighbours

Communication with neighbours is handled *in both directions at once* through a single construct, `nbr(f)`, which takes a flow `f` as a parameter. The local output of `f` will be automatically sent to neighbours. Instead, the output of the whole `nbr(f)` expression is an object `NeighborField[T]` collecting the values of `f` computed by all the neighbours. For instance

```
nbr(mid()) // or nbr(mid()).withoutSelf to exclude "self"
```

returns, in any device, the IDs of all its neighbours (including the device itself).

Sensors providing a value for each neighbour have dedicated syntax. They can be queried through construct `nbrSensor(name)`. For instance, the built-in function `nbrRange`, defined as follows:

```
def nbrRange(): Flow[NeighborField[Double]] =
  nbrSensor("nbrRange")
```

provides the neighbouring field of (estimated) distances to neighbours (how such a sensor works is an implementation detail—e.g., it may use GPS traces or Wi-Fi signal strength).

8.2.2.5 Branching

An expression `branch(c){t}{e}` evaluates and returns the value of expression `t` (resp. `e`) when `c` evaluates to `true` (resp. `false`). This enables a form of distributed branching, where devices that happen to execute `t` will not interact with those that executed `e` (and vice versa)—unlike `mux` in which a device "contributes" to both `t` and `e`. E.g., in a system split into red and blue devices, the expression:

```
branch(sensor("color") == "red"){
  nbr(constant(1)).sum // red nodes run this
} {
  nbr(constant(1)).sum // blue nodes run this
}
```

will yield in any device the number of neighbours *of the same kind*, neighbours that run the other sub-computation (despite those being the same) will not be considered. This concept is called *alignment* and is well-discussed, e.g., in [Aud+23].

Notice that, upon a change in the value sensed by `colour`, a device may dynamically switch branches and hence sub-computation domain and "aligned" neighbour set.

8.2.2.6 Lifting

Lifting enables flow combination: i.e., `lift(f1,f2,...,fN){g}` yields a flow obtained by applying `g(o1,o2,...,oN)` where `oi` is the output of flow `fi`. Lifting can also be applied on flows of `NeighborFields`, in which case the output is a flow of a `NeighborField` whose values are combined from the input `NeighborFields` neighbour-wise (runtime checks avoid combining flows with different domains). E.g.,

```
lift(nbr(mid(),nbrRange()){(nbrId,nbrDist) =>
  s"${nbrId} is at distance ${nbrDist} from me"}
```

yields locally to a device one string per neighbour reporting its ID and distance.

8.2.2.7 Looping (state evolution)

Construct `loop(init,ft)` evolves a piece of state (initially, `init`) by applying function `ft` mapping the previous state's flow to the next state's flow. For instance, the expression:

```
loop(0)(v => v + 1)
```

represents a computation counting from 0 onwards (ignoring overflow). How frequently does this counting progress? It depends on the implementation of `Context`, which provides a default throttling period. A different `loop` implementation may also accept a stream explicitly dictating the pace of the stateful computation (e.g., evolving state each time a button is pressed).

8.2.3 Paradigmatic Examples: Self-Healing Gradient & Channel

In this section, we cover the two fundamental self-organising behaviours already introduced in the Part I that will be exercised in the evaluation in Section 8.4. Firstly, we re-implement the *gradient* building block with the reactive extension, and we show how it can be used to implement a *channel* computation.

Example 1 (Self-Healing Gradient)**.** Figure 8.4a provides a representation of a gradient.

Multiple gradient computation algorithms exist [Aud+17] as previously outlined. A basic algorithm can be implemented in FRASP as follows.

Listing 8.1: Gradient implemented with FRASP

```
def gradient(source: Flow[Boolean]): Flow[Double] =
  loop(Double.PositiveInfinity) { g => {
    mux(source) {
      constant(0.0)
    } {
      lift(nbrRange(), nbr(g)){_ + _}
        .withoutSelf
        .min
    }
  }
```

The function takes the Boolean `source` flow as input, denoting whether the executing node is the source of the gradient or not. The external `loop` is used to progressively evolve the current gradient value `g` starting from an infinite value (as, initially, we do not know whether a source is reachable). Internally to the loop, we use `mux` to select one of two values: if the node is a source (i.e., `source` is true), then its gradient value is `0` (base case); otherwise, the gradient should be the minimum value among the neighbours' gradient values augmented by the distance (`nbrRange`) from that very neighbour. Construct `lift` is used to combine (using the sum, cf. `_+_`) the two flows `nbrRange` (distances to neighbours) and `nbr(g)` (neighbours' gradient values).

The following example showcases the *compositionality* of the programming model, namely the possibility of combining multiple self-organising behaviours to build a more complex self-organising behaviour.

Example 2 (Self-Healing Channel)**.** Therefore, an implementation in FRASP is as follows.

Listing 8.2: Channel implemented with FRASP

```
def broadcast[T](
  source: Flow[Boolean], value: Flow[T]
): Flow[T] =
  val broadcastResult = loop[(Double, Option[T])](
    (Double.PositiveInfinity, None)
  ) { d =>
    val x = value.map(0.0 -> Some(_))
    val y =
    mux(source) { value.map(0.0 -> Some(_)) } {
      val n = nbr(d)
      val distances = n.mapTwice(_._1)
      val values = n.mapTwice(_._2)
      val field = lift(distances, nbrRange(), values) {
        (ds, ra, va) => (ds + ra) -> va
      }
      field.withoutSelf
        .map(_.values.minByOption(_._1)
        .getOrElse((Double.PositiveInfinity,None)))
    }
  }
  lift(broadcastResult, value){_._2.getOrElse(_)}

def distanceBetween(
  source: Flow[Boolean], destination: Flow[Boolean]
): Flow[Double] =
  broadcast(source, gradient(destination))

def channel(
  source: Flow[Boolean],
  destination: Flow[Boolean],
  width: Double,
): Flow[Boolean] = lift(
  gradient(source),
  gradient(destination),
  distanceBetween(source, destination)
){
  (distSource, distDest, distBetween) =>
    distSource + distDest <= distBetween + width
}
```

This self-organising data structure can be implemented by leveraging two reactive `gradients` (one from the source and one from the destination—cf. Lines 33

and 34) and a `broadcast(v,s)` (which is a way to propagate a value `v` hop-by-hop from a source `s` outwards along the minimum paths of its gradient—indeed, a structure similar to the `gradient` implementation in Example 1) that supports the computation of `distanceBetween`. The `channel` depends on these three flows: i.e., the expression at Line 38 will be re-evaluated only upon a change of the output of (one of) these three sub-computations. For a graphical view of the local and distributed dependency graph, see Figure 8.1: the key idea is that, e.g., a local change in sensor `source` will not cause a re-computation of `gradient(destination)`.

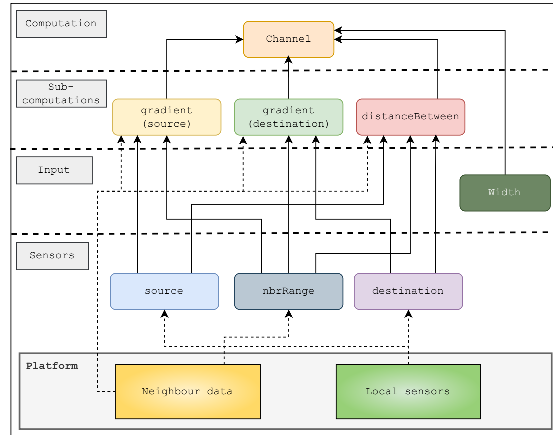
8.3 Implementation

This section briefly provides the implementation goals (Section 8.3.1), architectural design (Section 8.3.2), and implementation details (Section 8.3.3) of FRASP. Even though a complete description of the implementation is beyond the aims of this paper, this section is meant to illustrate that the prototype is technically sound, that we followed modern software engineering practices, and to provide general guidance for understanding the code organization of the provided artefact (see Footnote 1).

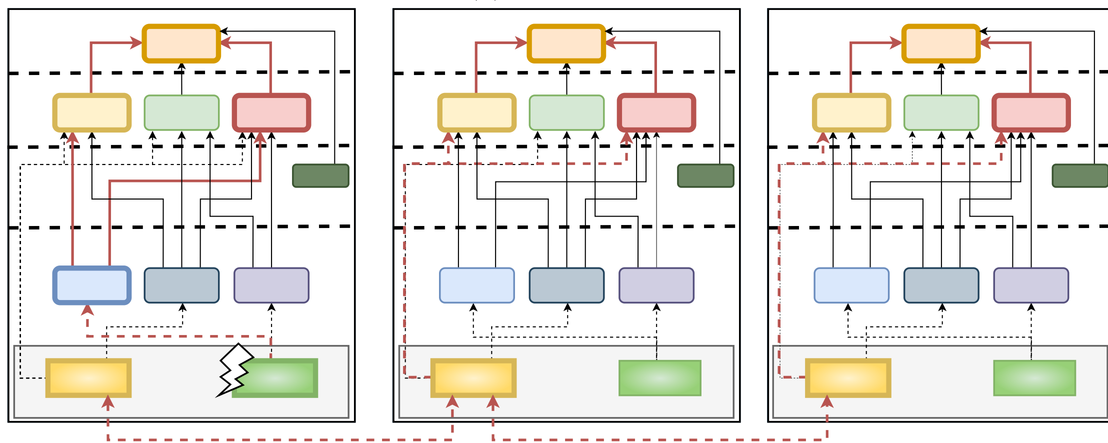
8.3.1 Goals

The high-level goal of this work is to provide a programming model that is expressive enough to allow developers to declaratively describe *self-organising* collective computations while decoupling and providing fine-grained control over their scheduling details. This vision can be summarized with the term *functional reactive self-organization*. Considering the system model introduced in Section 8.2.1, there are three main objectives to be pursued in order to accomplish the goal:

- *Re-compute only upon* relevant *changes in the context*: computations should occur reactively only when relevant changes are observed from the environment (i.e., sensing and neighbour data).
- *Avoid re-evaluation of unaffected sub-computations*: if a portion of the computation depends on data that did not change, it should not be re-evaluated.
- *Interact only upon relevant changes*: each device should avoid broadcasting an export that has not changed since the last one, with the direct consequence that no further message exchange should be required if a computation reaches a stable configuration.



(a) Node view.



(b) Distributed view (with neighbour dependencies).

Figure 8.1: The reactive dataflow graph corresponds to Example 2. Figure 8.1a provides the local view of the computation for a single node (where the layers denote different semantic kinds of dependencies), whereas Figure 8.1b shows the *distributed* dependency graph. The arrows denote dependencies. The dashed arrows denote dependencies based on platform-level scheduling and node interaction—e.g., a red block depends on changes corresponding to neighbours' red blocks and is communicated via message passing.

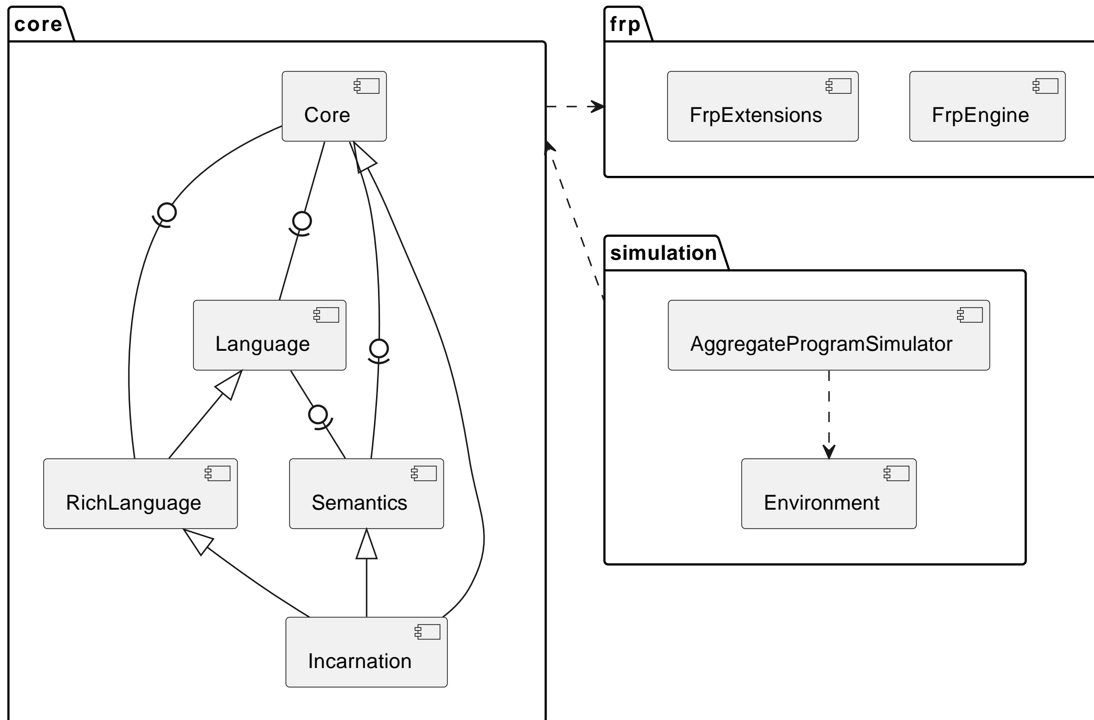


Figure 8.2: Architecture of FRASP.

8.3.2 Architecture

The architecture of the prototype is shown in Figure 8.2. The design is organised into three packages: `core`, which includes basic type definitions (`Core`) as well as the components for the DSL (`Language` for primitives and `RichLanguage` for other built-ins) and its "virtual machine" (`Semantics`), overall captured by an `Incarnation`; `frp`, which provides an interface to the FRP engine (`FrpEngine`), possibly also decoupling from the specific FRP library adopted, as well as extensions (`FrpExtensions`) useful for the definition of FRASP constructs; and `simulation`, which provides basic simulation support (for more advanced support, we also integrated FRASP into Alchemist—see Section 8.4).

8.3.3 Implementation details

FRASP has been implemented in Scala, using Sodium as FRP library [Bla16]. Scala is well-known for its suitability as a host for embedded DSLs [Art+15], and for aggregate computing embeddings as well [Aud+23]. The design of the FRASP DSL is detailed in Figure 8.3.

Following the system/execution model described in Section 8.2.1, we model

the input and output of a (sub-)program through an interface `Context`, providing access to local sensor data and neighbour data; and an interface `Export`, capturing outputs and data that must be shared with neighbours. In particular, an `Export` is modelled as a *tree* where each node is a `Slot` (corresponding to a particular language construct) with an associated value, and can be located through a *path* of slots—e.g., `S1/S2/S3` identifies a node in the export tree, where `S1` depends on `S2` which depends in turn on `S3` (so, a change in the output `S3` will cause the expression corresponding to `S2` to re-evaluate, and possibly `S1` in turn).

`Flow` is the type of a reactive (sub-)computation, which takes a `Context` (providing its inputs), a `Seq[Slot]` as path (indicating its position in the export tree), and returns `Cell` (i.e. a time-varying value—cf. Section 8.1.2) of `Export`. Each `Language` construct returns a `Flow`: therefore, the constructs do *not* immediately run upon evaluation, but rather an executable, reactive object denoting a computation graph whose nodes will execute as a response to change (cf. Figure 8.1). Access to neighbour-related data is mediated by a `NeighborField` abstraction, which is the same provided by constructs supporting interaction with neighbours, i.e., `nbr` and `nbrSensor` (cf. Section 8.2.2).

8.4 Evaluation

To evaluate the proposed approach, we prepared several publicly available and reproducible Alchemist simulations. In particular, we released FRASP[4] and integrated it into Alchemist.

8.4.1 Goals

The simulations are designed to evaluate the following:

G.1) *Correctness*: the reactive and round-based versions of the same algorithm should ultimately produce the same correct collective results.

G.2) *Efficiency*: The efficiency is measured in terms of:

(a) *messages*: the number of messages exchanged between devices;

(b) *time*: the time required to reach a stable output;

(c) *computation*: the number of sub-computation steps performed by each device.

[4] `https://central.sonatype.com/artifact/io.github.cric96/distributed-frp_3/0.1.3`

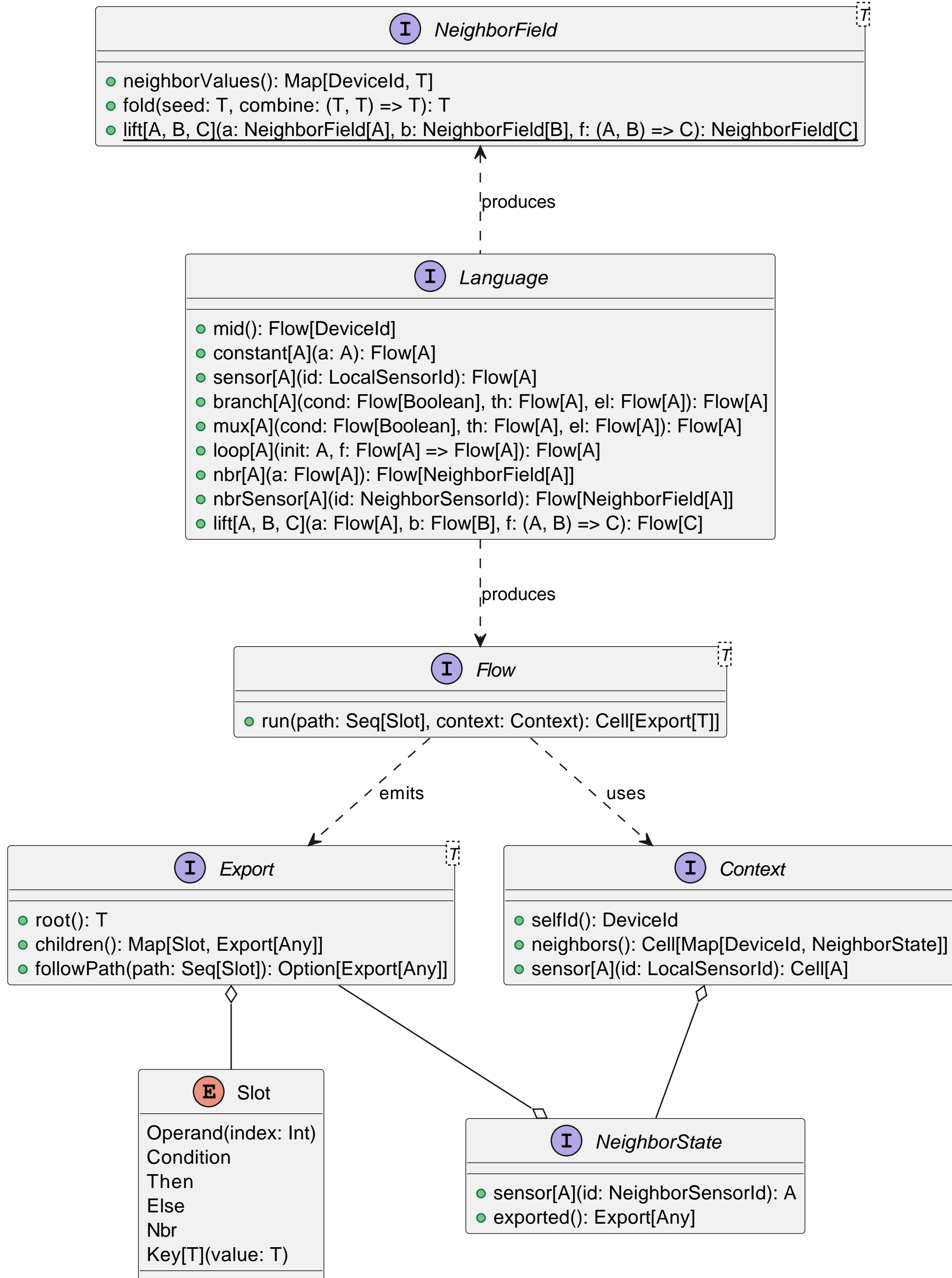
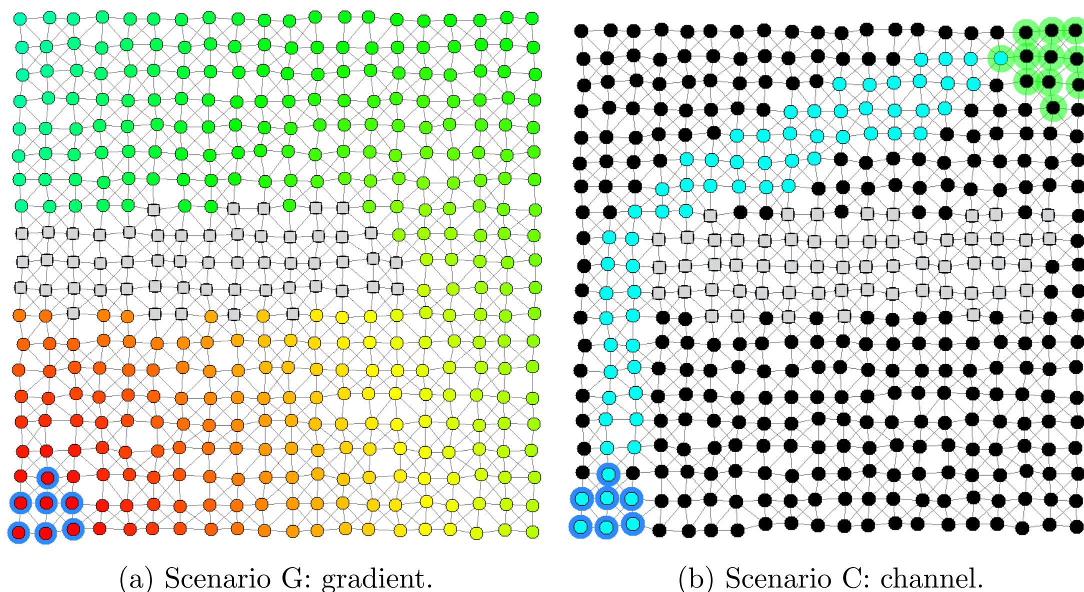


Figure 8.3: Design of FRASP DSL.



(a) Scenario G: gradient.

(b) Scenario C: channel.

Figure 8.4: Evaluation scenarios. The output of the program being evaluated is depicted in the inner circle. On the left, the hue varies depending on the gradient output, with redder colours indicating lower values. On the right, the channel between the source nodes (blue shadow) and destination (green shadow) is indicated in cyan. The grey nodes denote obstacles.

In particular, we expect the reactive version of the algorithm to be *more efficient* than the round-based one.

G.3) *Reactivity*: the programs should react to various sources of change (e.g., network topology, sensor data, dependent computations—cf. Section 8.2.1), avoiding re-evaluations when inputs do not change.

8.4.2 Experimental Setup

8.4.2.1 Common setup and parameters

The simulated system consists of 400 devices placed on a $100m \times 100m$ square and forming a slightly irregular grid (nodes' positions are generated for a regular grid and then randomly deviated). Each node has a mean distance to its neighbours of 5 meters and a communication radius of 7.5 meters. Messages sent by the nodes may be subject to a communication delay regulated by the parameter τ, which describes an exponential probabilistic function. Each simulation is characterized by the mode of execution of the aggregate program, which can be (i) *purely reactive*, (ii) *reactive with throttling*, or (iii) *round-based*. The pure reactive policy is evaluated

for every new message received from any neighbour. This policy is expected to converge quickly at the price of a significant overhead in the number of exchanged messages. The "reactive with throttling" policy is parametrized on the throttling frequency γ (i.e., the inverse of the period in which all events received in this interval are accumulated before emission). Compared to the purely reactive policy, throttling is expected to reduce message overhead at the expense of convergence time. Finally, the round-based policy is driven by a γ parameter describing how often each device will wake up and compute the round.

8.4.2.2 Scenarios

In the above setup, the two programs discussed in Section 8.2, i.e., the self-healing gradient (*scenario G*—cf. Figure 8.4a) and channel (*scenario C*—cf. Figure 8.4b), are executed for 300 simulated time units. The former represents a minimally complex self-organising behaviour, enabling the evaluation of basic dynamics, whereas the latter is representative of larger behaviours that can be defined as compositions of simpler ones, hence providing insights about what could happen when multiple reactive computations are combined.

For *scenario G*, a group of nodes (i.e., a $20m \times 20m$ area at the bottom left) is marked as a gradient source. Also, a set of nodes marked as obstacles is positioned at the centre in a $8m \times 2m$ area. To verify that the gradient can adapt to changes and assess the effects of continuous and frequent changes in the system, at simulated time $t = 150$ a cluster of nodes migrates from the lower left-hand side to the lower right-hand side of the area.

For *scenario C*, a set of nodes denoting the channel's destination are placed in the upper right in a $20m \times 20m$ area. Here, to verify reactivity to change, we switch the set of destination nodes at $t = 200$ from the upper-right corner to the upper-left corner. This injected change allows us to observe both the channel computation's reactivity and the sub-computations' evaluation. In fact, by only modifying the destination area, the gradient starting from the source (which is a sub-computation of the channel computation) should not be re-evaluated (as it would not change its collective output).

8.4.2.3 Metrics

The metrics extracted for this study are:

- *Total cumulative number of messages exchanged (up to time T)* – `#messages`: this is used to evaluate the communication overhead/efficiency (cf. goal G.2) and to inspect the communication dynamics of the reactive solution. It is computed as the sum of the number of messages sent by each node up to time T.

- *Average output value of the system* – `output (mean)`: for each time instant t, we extract the average value of the computational field produced by the system. This value will allow us to assess both correctness (cf. goal G.1) and time efficiency (when the programs converge to a certain stable value—cf. goal G.2). Also, this indicates the responsiveness of the system, as the output should vary with the introduction of changes in the system.

- *Total number of executions for program sub-computations* – `#evaluations`: this value was extracted in the channel program only (since it comprises multiple subprograms). For each sub-program, we calculated the number of times it has been evaluated up to time T as the sum of the number of evaluations of each node.

  This metric is useful for assessing goals G.2c and G.3.

8.4.2.4 Baselines

To establish baselines, we implemented round-based solution programs for the self-healing gradient (scenario G) and channel (scenario C) in ScaFi—see [Cas+22a]. The execution for the round-based solution is configured with an evaluation frequency of 1Hz: any device will evaluate the entire ScaFi program every second and then broadcast the resulting export to its neighbourhood (even if it did not change from the previous execution).

8.4.3 Results and Discussion

In this section, we will present the results obtained from the simulations, highlighting how the proposed model satisfies the goals elicited in Section 8.4.1. We run a total of 768 simulations by running 64 randomized instances (varying the random seed and the position of the nodes in the environment) for each one of the 12 *simulation configurations* obtained by a different combination of *(i)* execution mode, *(ii)* throttling period, and *(iii)* scenario program. Our results are presented in Figure 8.5 and Figure 8.6.

8.4.3.1 Correctness (goal G.1)

Consider Figure 8.5b and Figure 8.5d: we observe that in all cases, especially in the reactive and throttling executions, the program output converges to the same collective result. Pure reactive policies have less noise (as variations are reduced) but tend to converge to the same result on average.

8.4.3.2 Efficiency (goal G.2)

From Figure 8.5, we can see how the reactive computation is more efficient than the round-based one (cf. goal G.2) in the sense of communication/computation efficiency (Figures 8.5a and 8.5c) and time efficiency (Figure 8.5d). Moreover, as expected, throttling can further improve the efficiency of the reactive solution.

Indeed, starting with the messages metric, we observe that the number of messages exchanged in pure reactive policies to reach a stable output is greater than the throttled counterpart. This result is expected: in the purely reactive model each message different from the previous one causes a program re-evaluation and subsequent message sending. Analysing the policies with throttling, we notice that the higher the throttling period, the lower the number of messages sent, and in particular, the more the policy tends towards the round-based one. In the gradient scenario, for instance, the throttle mode achieves convergence by sending six times fewer messages than the round-based one. Despite consuming more, purely reactive policies are still more efficient than round-based ones, sending approximately 40% fewer messages. However, it is also noteworthy that, in the case of continuous variations, such as the migration of nodes in the gradient scenario, the number of messages grows linearly in both reactive and round-based modes. Efficiency could be improved by approximating the output value not to send every message for every update, i.e., to regulate reactivity according to some threshold for "significant change".

Moving on to converge time, we immediately note that reactive policies are the most efficient because they expand changes throughout the system as soon as they occur, avoiding delays due to waiting for a new evaluation round. However, in the case of frequent environmental change, one may have a higher message consumption than round-based policies, leading to higher energy consumption.

Observing throttling policies, we note that the higher the throttling period, the higher the convergence time, as the system will take longer to react to changes. In the worst case (i.e., when the throttling period is equal to the evaluation time of round-based policy), the convergence time is approximately the same. However, convergence time shortens with the decrease in the throttling period, getting performance close to the purely reactive policy but with fewer messages exchanged. Thus, *the proposed model balances communication cost (the number of messages exchanged) and performance (time required to converge to the expected output), depending on the application's needs.* Moreover, the relationship between the throttling period and the number of messages sent is not linear (unlike the convergence time). In fact, halving the period (0.5 seconds instead of 1 second) also halves the convergence time, yet the number of messages exchanged is approximately the same.

Concerning reactivity (goal G.3), we notice how the reactive solutions effec-

tively "follow" environmental changes. This can be observed from Figure 8.5a and Figure 8.5c: when there is no change to react to, no program evaluation is performed, resulting in no new messages being sent (cf. the intervals where the message count does not increase). Moreover, as the environmental conditions vary, the program is re-evaluated, as observed at the moment of variation (at $t = 150$ in the case of the gradient and at $t = 200$ in the case of the channel).

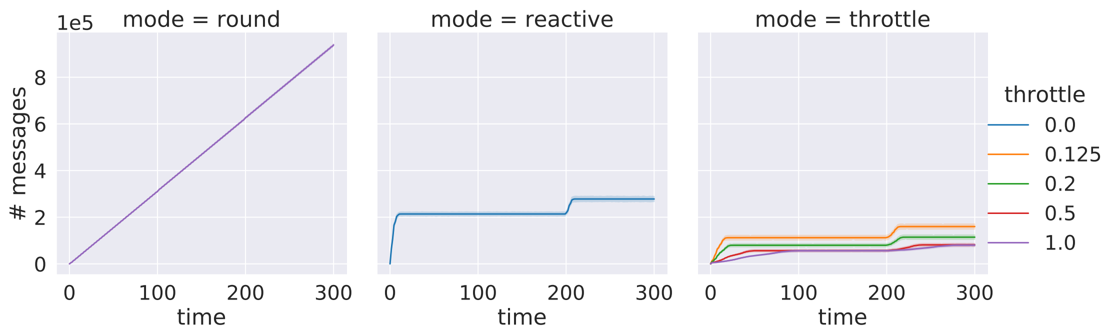
Finally, we show how the model promotes *fine-grained reactivity*. Figure 8.6 shows how FRASP enables a program to re-compute required sub-computations *only*.

Indeed, at $t = 200$, only the channel's target changes, and thus the source gradient is not re-computed (it would be pointless since its inputs did not change). Also, this execution occurs only where needed, resulting in a "wave" of re-computations from the source to the destination (cf. the larger red circles in Figure 8.6).

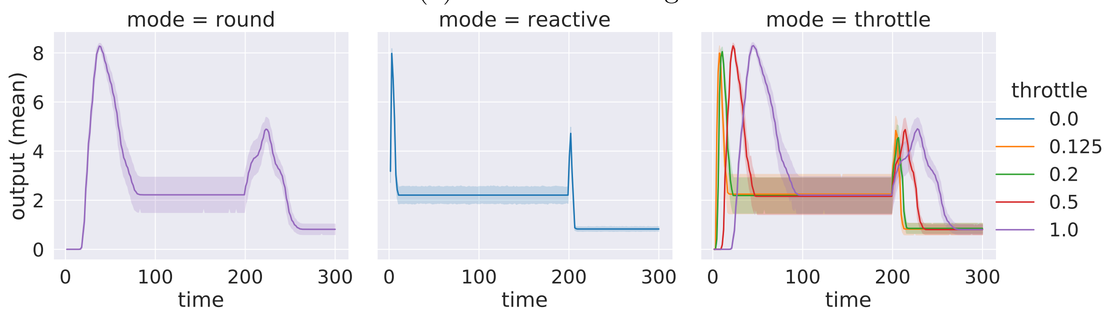
8.5 Final Remarks

This chapter proposes FRASP, a *functional reactive macroprogramming model for expressing self-organising behaviour*. The language is designed by taking inspiration from the aggregate programming approach and can be seen as an extension of it. As experimentally verified, FRASP allows *tunable, fine-grained reactivity*, enabling increased communication and time efficiency w.r.t. proactive models. Indeed, in FRASP, a distributed self-organising computation turns into a distributed computation dependency graph, where distinct sub-computations may execute independently depending on whether their context has changed. This represents a seminal advancement in the field of CPSW engineering, as it enables both the expression of self-organizing behaviours– crucial for the development of robust collective applications– and fine-grained control over computational scheduling—key for optimizing the efficiency of these applications.

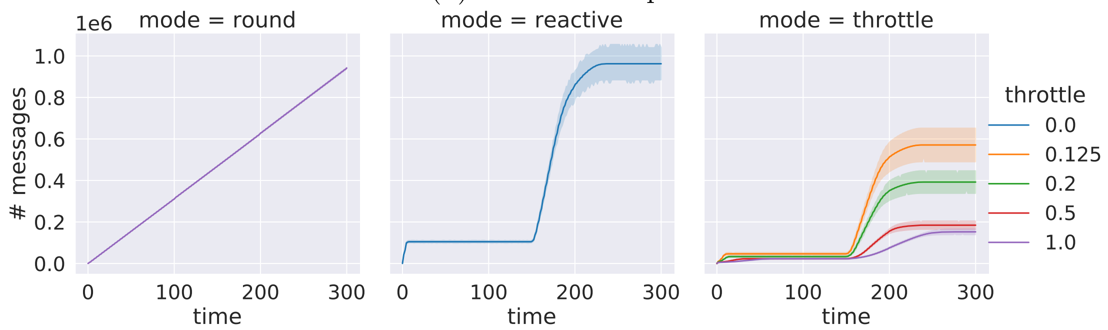
These chapters serve as a bridge between the complex problem domain of CPSW and practical solution engineering. However, the question of how to effectively deploy such intricate behaviours within contemporary complex IT infrastructures remains unresolved. In the final chapter, we explore a potential approach to address this challenge, melding the innovative aspects of pulverized architecture with multi-tier programming.



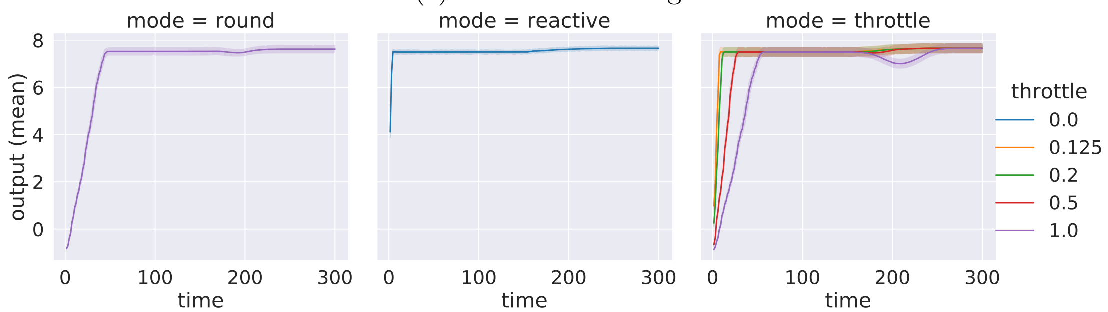
(a) Channel: messages.



(b) Channel: output.



(c) Gradient: messages.



(d) Gradient: output.

Figure 8.5: Simulation results of FRASP simulations.

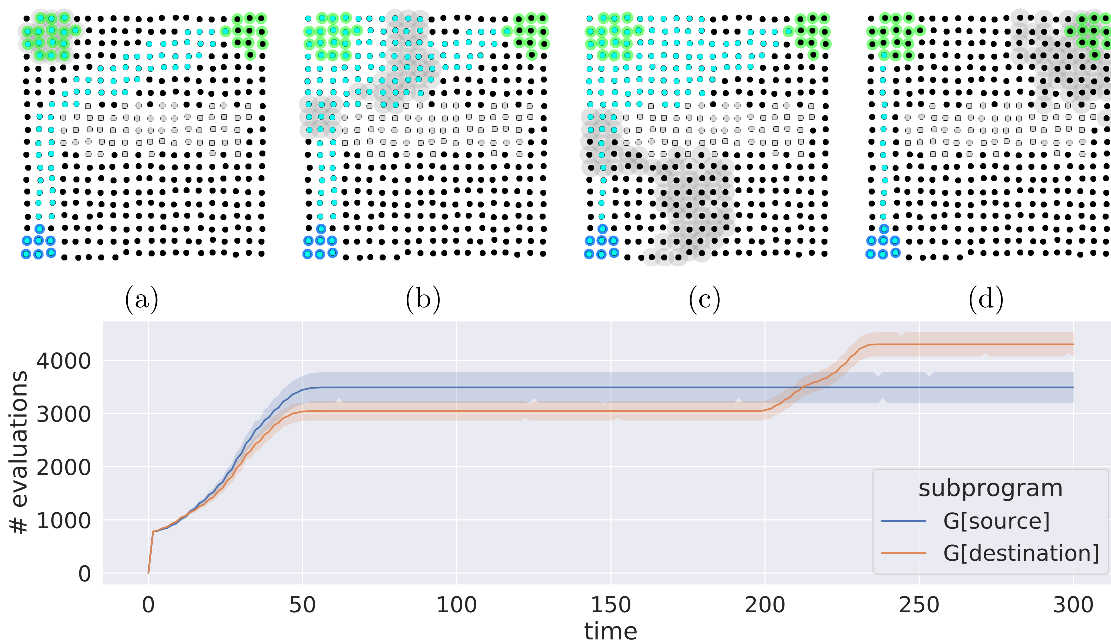


Figure 8.6: behaviour of the channel in response to changes in a destination

Chapter 9

Platform: Deployment of Cyber-Physical Swarms applications

How can we deploy an aggregate computing application in a large variety of network architectures?
Does the deployment strategy affect the functional behaviour of the application?
How can we ensure that the deployment strategy is correct and safe?
– **RQ1, RQ4**

Contents

9.1 Background 176
9.1.1 Pulverized aggregate computing 176
9.1.2 Multi-tier programming and ScalaLoci 177
9.2 Multi-tier pulverised aggregate computing 179
9.2.1 Pulverized architecture in ScalaLoci 180
9.2.2 Definition of deployment kinds 180
9.2.3 Integration with aggregate programming 183
9.3 Implications 184
9.4 Final remarks 186

CPSW are typically required to act as coordinated collectives and to be able to adapt to dynamic environmental conditions and inputs. As discussed in Chapter 3 reliance on a centralized device is not allowed, and the system is expected to reach global goals based on the coordinated (inter-)actions of the individual entities that compose the system. Aggregate computing foster the adoption of *global-to-local* techniques, by which the behaviour of the ensemble of devices is designed top-down, and interactions among devices (i.e., protocols) are generated automatically and implicitly. However, the generated interaction scheme depends on assumptions about how devices communicate with each other; in other words, the approach often dictates how the network should be structured. In turn, this creates a tension between the network structure the language reasons upon and the actual way devices communicate. Since in CPSWs the challenging case in which no controllers exist must typically be supported, a purely *peer-to-peer (P2P)* network is usually considered the paradigmatic setup. However, real-world networks are usually structured hierarchically, and the ability to target multiple infrastructural setups can help to achieve non-functional benefits. Therefore, a contemporary approach designed to facilitate agile deployment of CPSW should be versatile enough to accommodate multiple network architectures. In this direction, *Pulverization* [Cas+20b] is an approach proposed for aggregate computing (but in principle applicable to other frameworks) to neatly separate behavioural and deployment concerns. In short, it decomposes the concept of a *logical device*, which is the target for which the behaviour is programmed, into micro-components that can be deployed independently and whose internal communication protocol is defined at deployment time. This technique de facto relieves the behaviour designer of the duty to consider multiple possible deployments in different networks, allowing them to write the behaviour for the most generic case, and have the program *functionally* behave as designed regardless of the actual final deployment.

However, pulverization does not directly provide ways for *specifying* and *deploying* components in a safe and meaningful fashion: it proposes a methodology to cleanly separate behavioural and deployment concerns that need to be addressed at some point. The definition of the deployment strategy and its execution is thus a very relevant and challenging engineering issue on its own: ideally, such a specification should be declarative and possibly guided and checked by static analysis to lower the risk of failures at runtime.

This chapter discusses how a pulverized system can be deployed and executed on multiple different network structures, by leveraging a recent approach known as *multi-tier programming* [WWS20]. In multi-tier programming, a distributed system is *declaratively* described, in terms of components and admissible interactions *in a single code base*. In particular, in type-level multi-tier programming, the specification leverages the type system of the language to ensure the correctness

and coherence of the architecture; moreover, it also relieves the developer from low-level concerns by offloading to the compiler the responsibility of breaking the computation into deployment units enforcing the contract defined in the code.

The integration of these two paradigms offers a promising strategy for designing and implementing CPSWs that operates independently of the underlying infrastructure. In this chapter, we explore this approach by combining ScaFi and ScalaLoci—a Scala-based framework for multitier programming.

9.1 Background

9.1.1 Pulverized aggregate computing

At the core of pulverization [Cas+20b] is the idea that the functional behaviour of a distributed application is fundamentally orthogonal to the actual deployment of the services that compose it. Thus, through a classic *divide-and-conquer* approach, in a pulverized system, any *logical device* (of the many composing the CPSW) is broken down into five *components* acting as *units of deployment*: 1. *Sensors (***S***)*, encapsulating the ability to retrieve information from the environment; 2. *Actuators (***A***)*, responsible for acting upon the environment; 3. *State (***K***)*, providing persistence of knowledge; 4. *Behaviour (***B***)*, modelling the actual execution of the application business logic; and 5. *Communication (***C***)*, which provides means to interact with other logical devices. These pulverized components can be deployed to different physical nodes of the network: as far as they can communicate with each other and the target execution protocol is respected, the functionality should not be affected. Then, a concrete development approach will expose abstractions with a well-defined mapping to such a partitioning schema. So, an application designer can focus on functional requirements while delaying all the deployment and communication concerns (which may well affect non-functional properties of the system) to a later moment.

This strategy is especially well-suited to adapt approaches designed to work with a flat (non-layered) network structure (e.g., peer-to-peer, mesh, and ad-hoc networks) to arbitrary network architectures—to exploit a broader range of deployments, e.g. for efficiency or reliability. Consider, for instance, the simple case of Figure 9.2a: there is a 1:1 mapping between logical and physical devices, and direct communication among devices (actually, among their **C** pulverized components) must be possible. This is typically not the case in many Internet applications, however: let us consider the case in which the same application should be deployed in an IoT scenario where end devices are *thin*, equipped with sensors and actuators, but battery-powered and equipped with a microcontroller with minimal computational capabilities (for instance, LoRaWAN or Sigfox motes [Mek+18]).

These devices cannot host the actual computation of the program (component **B**), which must necessarily be offloaded to the edge of the cloud. This change in the deployment would typically imply a re-writing of the functional logic of the program, as end devices cannot be considered computation-capable nodes any more. Instead, with pulverization, they retain their existence as logical devices with some of their pulverized components hosted on different physical nodes as depicted in Figure 9.2c. Crucially, this makes the original application work on a different network architecture without any functional logic changes.

Aggregate computing is a naturally pulverizable approach: its semantics can be expressed as a purely functional manipulation of state, messages, and sensor readings [Vir+19], providing a straightforward mapping into pulverized components. Indeed, initial experiments [Cas+20b] showed that aggregate programs deployed on pulverized infrastructures retain their original functional behaviour on different deployments. Nevertheless, pulverization is not a silver bullet: the approach is fundamentally an engineering pattern to encapsulate a non-functional concern (network structure and deployment), allowing for the business logic to work across deployments, but it does not specify *how* a pulverized architecture should be described and verified so that it can be operated correctly at runtime.

9.1.2 Multi-tier programming and ScalaLoci

The concrete architecture of a distributed system is usually *multi-tier*, i.e., it comprises multiple layers, each one encapsulating some specific functional concern (e.g. data management, application and presentation logic, etc.) each physically separated from the others. Historically, distinct tiers and crosscutting functionalities that belong to multiple tiers are developed into several compilation units (often using different programming languages), raising development and maintenance costs.

A recent trend in trying to tackle these issues is *multi-tier programming* [WWS20], by which a distributed architecture is defined in a single compilation unit with a single language. Once the program is declaratively specified, the compiler (or the runtime, depending on the language of choice) is responsible for splitting the computation among different peers. Depending on the specific multi-tier programming language, different kinds of constraints may be imposed. For instance, in Links [Coo+06], applications must follow a client-server architecture, while other languages allow for more freedom of choice.

One interesting language that lets the designer specify arbitrary deployments is ScalaLoci [WKS18; WS19; WS20], a type-safe multi-tier language hosted in Scala language. The structure of a ScalaLoci application is defined through *peers* and *ties*. Peers abstract over locations and represent the components of an application, whereas ties define the connections between peers. Only tied peers can communicate with each other.

The following code depicts a simple controller-worker architecture. Annotation `@multitier` denotes the `BookingApp` as a ScalaLoci object.

```
/* Defines an application with the peers 'Controller' and
 * and 'Worker' and a 1:n connection between them */
@multitier object BookingApp {
  @peer type Controller <: {
    type Tie <: Multiple[Worker]
  }
  @peer type Worker <: {
    type Tie <: Single[Controller]
  }
  on[Controller]{ print("I am a Controller") }
  on[Worker]{ print("I am a Worker") }
}
```

A declared `@peer` type can have multiple instances that execute the peer's logic, e.g., multiple worker instances. In this example, the logic is replaced with simple prints. An instance of the controller peer may connect to multiple workers, whereas a worker instance is tied to one controller. The sample compiles two executables representing the controller and the worker, whose instances can be deployed and executed on different physical nodes.

```
/* accessible for workers. */
val requests: Event[Request] on Controller = placed {...}
// Name of the worker @Worker accessible for Controller.
val name: String on Worker = placed {...}
/* not accesible for workers. */
val tokens : Local[Map[Long]] on Controller =
  placed {...}
/* Access allowed: Worker observes events
   emitted on Controller. */
on[Worker]{ requests.asLocal.observe{...} }
// Error: no access to tokens outside of the Controller.
on[Worker]{ tokens.asLocal.observe{...} }
```

Asynchronous multi-tier reactives like signals and events are used to compose non-blocking data flows that span across multiple peers. Data from remote peers are accessed using ScalaLoci's `.asLocal` expression variants, and the visibility of placement types for remote peers can be regulated. A `@multitier` module can capture the controller-worker schema:

```
@multitier trait ControllerWorker[T] {
  @peer type Controller <: {
    type Tie <: Multiple[Worker]
```

```
  }
  @peer type Worker <: {
    type Tie <: Single[Controller]
  }
  def run(task: Task[T]): Future[T] on Controller =
    // run task on some selected worker
    on(selectWorker()) // (`selectWorker` is left out)
      .run.capture(task) { task.process() }.asLocal
}
```

The run method has return type as the placement type Future[T] on Controller[1], effectively placing run on the Controller peer. The Task type is parametrized over the type T of the value, which a task produces after execution. Running a remote task remotely results in a Future to account for processing time network delays and potential failures. The remote block is executed on the worker, which starts processing the task. The remote result is transferred back to the controller as Future[T] using asLocal. A single worker instance in a pool of workers is selected for processing the task via the selectWorker method.

The module can be used to implement an application where a server offloads work to the connected clients. In the following code, we specialize the clients to be workers and the server to be a controller:

```
@multitier trait VolunteerProcessing {
  val m: ControllerWorker[Int] // ref to another module
  // augmenting the peers in this module
  @peer type Client <: m.Worker
  // with the controller/worker functionality
  @peer type Server <: m.Controller
  on[Server] { m.run(new Task()) }
}
```

9.2 Multi-tier pulverised aggregate computing

The contribution of this work is an architecture for multi-tiered deployment strategies in pulverized systems, along with a prototypical implementation using aggregate programming and ScalaLoci. Using multi-tier abstractions, we:

1. map the overall logical system into a multi-tiered module, building the concept of a pulverized device into ScalaLoci (see Figure 9.1b);

[1]Scala enables infix use of binary type constructors; i.e., A on B refers to the same type as on[A,B] .

2. define the functions associated with each pulverized component;

3. characterize the possible kinds of network nodes (e.g., cloud, edge, thin-end device);

4. decide the network structure in terms of possible connections among network node kinds;

5. detail the deployment by assigning each pulverized component to a network node kind.

Ultimately, this architectural design allows us to specify functional behaviour independently of deployment (via pulverized aggregate programming), then declaratively define multiple deployment schemes and their related communication constraints (thanks to multi-tier programming), and finally, statically enforce the respect of the expressed constraints (as a consequence of the robust type programming system introduced by ScalaLoci).

9.2.1 Pulverized architecture in ScalaLoci

As a first step, we need to formalize what a pulverized architecture is in ScalaLoci, by defining all the pulverized components and binding them together into the concept of the logic node. Figure 9.1 shows a possible ScalaLoci implementation (Figure 9.1b) of a pulverized device (Figure 9.1a): `LNode` represents the logical device, `LogicalSystem` encloses the concept of the pulverized system into a multi-tier module. The logical device and all its pulverized components are mapped on abstract peers.

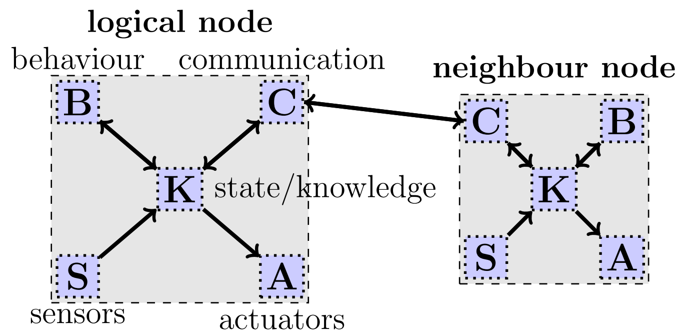
Once all components are modelled, their contract must be specified to characterize them and define their behaviour. This is done by *placing* the available computations on the components that will effectively host them. For instance, if our system has the notion of `Sensor[V]`, representing a generic sensor that upon access returns values of type `V`, we can enforce the requirement that the `SensorComponent` must be able to read values from sensors via something like:

```
def sense[V](id: SensorID): V on SensorComponent = ...
```

This strategy decouples the *structural* definition of components participating in the system from their *behavioural* specification.

9.2.2 Definition of deployment kinds

Once the definition of components is complete, we can begin describing the actual deployments. These can be expressed rather concisely with the proposed

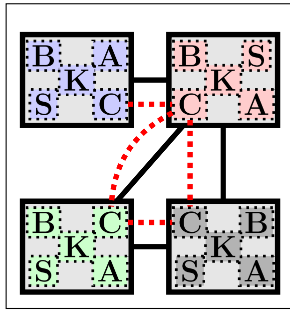


(a) A pulverized logical device split into sub-components, and one of its neighbours.

```
@multitier trait LogicalSystem {
  // A logical node, connected to other logical nodes
  @peer type LNode <: { type Tie <: Multiple[LNode] }
}
// Partitioning of a logical node into sub-components
@multitier trait PulverisedSystem extends LogicalSystem {
  @peer type SensorComponent <: LNode // $\LSens$
  @peer type ActuatorComponent <: LNode // $\LAct$
  @peer type StateComponent <: LNode // $\LState$
  @peer type BehaviourComponent <: LNode // $\LComp$
  @peer type CommunicationComponent <: LNode // $\LComm$
}
```

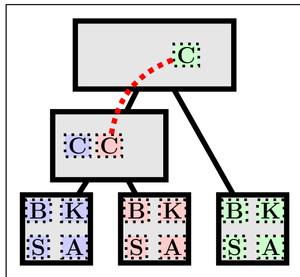
(b) ScalaLoci code describing a pulverized logical system. (See Section 9.2.3 for more details.)

Figure 9.1: Pulverization model and corresponding ScalaLoci specification.



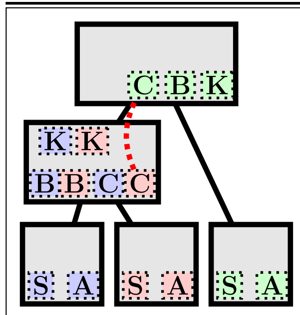

```
@multitier object P2P extends PulverisedSystem {
  /* Definition of device
  kinds and possible network connections */
  @peer type Node <: { type Tie <: Multiple[Node] }
  @peer type SensorComponent <: Node
  // Pulverised component allocation on devices
  @peer type ActuatorComponent <: Node
  @peer type StateComponent <: Node
  @peer type BehaviourComponent <: Node
  @peer type CommunicationComponent <: Node
}
```

(a) Peer-to-peer: 1-to-1 mapping between logical and physical devices.




```
@multitier object BrokerBased extends PulverisedSystem {
  // Definition of device kinds
  // and possible network connections
  @peer type Node <: { type Tie <: Single[Broker] }
  @peer type Broker <: {
    type Tie <: Multiple[Node] with Multiple[Broker]
  }
  @peer type SensorComponent <: Node
  @peer type ActuatorComponent <: Node
  @peer type StateComponent <: Node
  @peer type BehaviourComponent <: Node
  @peer type CommunicationComponent <: Broker
}
```

(b) Multi-broker: the communication is offloaded in part to the edge, and in part to the cloud.




```
@multitier object IoTSystem extends PulverisedSystem {
  // Definition of device kinds and possible network connections
  @peer type Thin <: { type Tie <: Multiple[Thick] }
  @peer type Thick <: { type Tie <: Multiple[Thick] with Multiple[Thin] }
  // Pulverised component allocation on devices
  @peer type SensorComponent <: Thin
  @peer type ActuatorComponent <: Thin
  @peer type StateComponent <: Thick
  @peer type BehaviourComponent <: Thick
  @peer type CommunicationComponent <: Thick
}
```

(c) IoT with thin clients: end devices only host sensors and actuators, other components are offloaded either to the edge or the cloud.

Figure 9.2: Examples of pulverized architectures. Thick boxes represent physical devices, dashed boxes represent pulverized components, and different logical devices are identified by colour (red, green, and blue). A pulverized component is hosted on the physical device in which it is contained. Communication among different logical devices that imply communication among physical devices is depicted with a dashed red line.

design, as depicted in Figure 9.2, where we show three possible definitions of very different architectures. In Figure 9.2a, we define a system where logical and physical devices coincide. This structure is typical of opportunistic network structures (P2P overlays, tactical networks, etc.). Figure 9.2b, shows a hybrid edge-cloud system supporting the computation of *thick* end devices (e.g., smartphones). The infrastructure hosts the communication components, de facto enabling network communication among end devices (this is a typical situation in usual Wi-Fi networks, where end devices are "hidden" a router performing network address translation). A practical example of this architecture could be a multi-broker MQTT system, with brokers deployed either on the edge (for better performance with closely located devices) or on the cloud. Finally, in Figure 9.2c, we replicate a similar system, but with *thin* end devices. Namely, end devices do not possess enough computational capacity to host their associated computation and thus need to operate as remote sensors and offload all calculations to an external device. This situation is typical of WAN sensing networks (e.g., LoRaWAN), where end devices are equipped with minimal memory and very low-power microcontrollers and are expected to run on battery for years. To summarize, different network architectures can be specified by following two steps: 1. definition of the physical devices involved in the architecture and how they are *tied* together; and 2. allocation of the pulverized components on the kinds of devices that can host them.

The resulting system can then be instanced by selecting a communication protocol and a serialization framework. For example, in the following snippet, we show how this could be done for the system in Figure 9.2b, assuming communication via TCP and serialization via the uPickle library.

```
import loci.serializer.upickle._ // Serialization logic
import loci.communicator.tcp._ // Communication protocol
object Broker extends App { // Peer instatiation
  val tie = listen[BrokerBased.Peer](TCP(port))
  multitier.start(new Instance[BrokerBased.Broker](tie))
}
object Peer extends App {
  val tie = connect[BrokerBased.Broker](TCP(host, port))
  multitier.start(new Instance[BrokerBased.Node](tie))
}
```

9.2.3 Integration with aggregate programming

The design described so far is entirely independent of the specific aggregate programming language of choice: due to pulverization, the way the logic is expressed

only concerns the behavioural component (**B**). As reference aggregate computing library we picked ScaFi for our prototype, mainly because it shares the language of choice with ScalaLoci, and thus it could be the foundation stone of a unified framework living in the Scala ecosystem. A full account of ScaFi can be found in Part I.

To perform a collective computation, ScaFi requires to define an `AggregateProgram` (i.e. an object containing the aggregate application logic) and a `Context` (i.e., the set of information required to evaluate an `AggregateProgram`, such as the previous state, sensors' data, and messages received from neighbours). ScaFi's `Contexts` in a pulverized architecture are embedded in the `State` component. Consequently, the glue code required to execute ScaFi aggregate code over a pulverized network is minimal:

```
def compute(
  deviceIdentifier: Id,
  state: State
): State on BehaviourComponent = {
  val context = new ContextImpl(
    deviceIdentifier,
    export = state.exports,
    localSensor = state.sensors,
    neighbourSensor = state.neighbourSensor
  )
  val program : AggregateProgram = ... // business logic
  // actual execution; returns the new State
  program.round(context)
}
```

9.3 Implications

The construction of a pulverized platform for aggregate computing via multi-tier programming introduces a plethora of intriguing applications, each with its unique impact and significance. In this section, we delve deeper into these applications, examining their technical intricacies and their broader implications for both academia and industry.

Programmability and Compile-Time Safety in Deployment Architectures for Pulverized Systems The use of type-annotated definitions in the deployment of a pulverized system confers robustness and reliability to the architectural integrity of the system. This approach facilitates static consistency checks

between the planned architecture and its deployed instantiation, enabling the compiler to enforce constraints. As a result, unauthorized access to data that is not within the scope of a specific component is precluded by design. This dramatically enhances system security and diminishes runtime errors.

From an engineering standpoint, the focus then squarely shifts to addressing the functional aspects of the application. In this context, the functional aspect refers to crafting the core logic of an aggregate computing program. We posit that this approach can be subsumed under a broader research paradigm aimed at modularizing functional and non-functional concerns. The latter could then be managed using specialized techniques and languages tailored for those specific challenges.

Opportunistic Deployment and Dynamic Reconfiguration of Pulverized Systems The inherent flexibility of the pulverized architecture enables dynamic adaptability in application deployment. Essentially, it is conceivable to reposition components of a logical device across multiple physical nodes dynamically. For instance, a **B** component could be offloaded to the cloud to conserve battery life on the original device. While this concept is conceptually supported by the pulverization methodology, its practical implementation is currently limited due to ScalaLoci's focus on static data placement.

Nonetheless, our research opens the door to future developments that could extend existing languages to support dynamic data placement between peers. Such an extension would not only augment system flexibility but also maintain type safety in the system specification.

Incorporation of Placement Types in aggregate programming As it stands, ScalaLoci and ScaFi have been designed to function in tandem in such a way that the aggregate program remains blissfully unaware of its multi-tier deployment architecture. However, another avenue for exploration is how placement types, along with other novelties introduced by ScalaLoci's unique take on multi-tier programming, could be strategically utilized within aggregate computing systems.

At present, the specific implications and potential benefits of manipulating placement types at the level of aggregate computing are not yet fully understood. Nonetheless, we see emerging opportunities for developing adaptive networked systems capable of dynamic evolution. These preliminary insights suggest a fertile ground for further research and development in this area.

9.4 Final remarks

In this chapter, we introduce a methodology for bridging fragmented architectures with the verified deployment of aggregate systems. We utilize multitier programming to enhance declaratively, expressiveness, and safety. Specifically, we demonstrate how to define a fragmented architecture using ScalaLoci and then map it to an actual deployment.

Additionally, we outline how aggregate computations can be seamlessly integrated into the existing infrastructure. This is achieved by sketching the implementation of the system's behavioural aspects using the ScaFi aggregate computing toolkit.

Given the vast design possibilities in this context, we posit that the synergy between multitier programming and fragmented aggregate programming presents a compelling strategy for designing and implementing CPSW. This approach allows for independent operation from the underlying infrastructure while maintaining the integrity of the business logic.

References

[Abd+21] Mohamed Abdelkader et al. “Aerial Swarms: Recent Applications and Challenges”. In: *Current Robotics Reports* 2.3 (July 2021), pp. 309–320. DOI: 10.1007/s43154-021-00063-4. URL: https://doi.org/10.1007/s43154-021-00063-4.

[Alb+22] Daniel Albiero et al. “Swarm robots in mechanized agricultural operations: A review about challenges for research”. In: *Comput. Electron. Agric.* 193 (2022), p. 106608. DOI: 10.1016/j.compag.2021.106608. URL: https://doi.org/10.1016/j.compag.2021.106608.

[AP22] Gianluca Aguzzi and Danilo Pianini. *cric96/experiment-2022-ieee-decentralised-system: 1.0.1.* 2022. DOI: 10.5281/ZENODO.6477039. URL: https://zenodo.org/record/6477039.

[Art+15] Cyrille Artho et al. “Domain-Specific Languages with Scala”. In: *ICFEM.* Vol. 9407. LNCS. Springer, 2015, pp. 1–16. DOI: 10.1007/978-3-319-25423-4_1.

[Ash+07] Michael P. Ashley-Rollman et al. “Meld: A declarative approach to programming ensembles”. In: *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems.* IEEE, 2007, pp. 2794–2800. DOI: 10.1109/IROS.2007.4399480. URL: https://doi.org/10.1109/IROS.2007.4399480.

[Aud+17] Giorgio Audrito et al. “Compositional Blocks for Optimal Self-Healing Gradients”. In: *SASO.* IEEE. 2017, pp. 91–100. DOI: 10.1109/SASO.2017.18.

[Aud+22] Giorgio Audrito et al. “Functional Programming for Distributed Systems with XC”. In: *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany.* Ed. by Karim Ali and Jan Vitek. Vol. 222. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 20:1–20:28. DOI: 10.4230/LIPIcs.ECOOP.2022.20.

[Aud+23] Giorgio Audrito et al. "Computation Against a Neighbour: Addressing Large-Scale Distribution and Adaptivity with Functional Programming and Scala". In: *Log. Methods Comput. Sci.* 19.1 (2023). DOI: 10.46298/lmcs-19(1:6)2023.

[Aud20] Giorgio Audrito. "FCPP: an efficient and extensible Field Calculus framework". In: *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020, Washington, DC, USA, August 17-21, 2020.* IEEE, 2020, pp. 153–159. DOI: 10.1109/ACSOS49614.2020.00037. URL: https://doi.org/10.1109/ACSOS49614.2020.00037.

[Bai+13] Engineer Bainomugisha et al. "A survey on reactive programming". In: *ACM Comput. Surv.* 45.4 (2013), 52:1–52:34. DOI: 10.1145/2501654.2501666.

[Bal+13] David Ball et al. "Robotics for Sustainable Broad-Acre Agriculture". In: *Field and Service Robotics - Results of the 9th International Conference, December 9-11, 2013, Brisbane, Australia.* Ed. by Luis Mejías Alvarez, Peter I. Corke, and Jonathan M. Roberts. Vol. 105. Springer Tracts in Advanced Robotics. Springer, 2013, pp. 439–453. DOI: 10.1007/978-3-319-07488-7_30.

[BBM10] Jonathan Bachrach, Jacob Beal, and James McLurkin. "Composable continuous-space programs for robotic swarms". In: *Neural Comput. Appl.* 19.6 (2010), pp. 825–847. DOI: 10.1007/s00521-010-0382-8.

[BDT99] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence - From Natural to Artificial Systems.* Studies in the sciences of complexity. Oxford University Press, 1999. ISBN: 978-0-19-513159-8.

[Bea+13] Jacob Beal et al. "Organizing the Aggregate: Languages for Spatial Computing". In: *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments.* IGI Global, 2013. Chap. 16, pp. 436–501. ISBN: 978-1-4666-2092-6. DOI: 10.4018/978-1-4666-2092-6.ch016.

[Bes+19] Graeme Best et al. "Dec-MCTS: Decentralized planning for multi-robot active perception". In: *Int. J. Robotics Res.* 38.2-3 (2019). DOI: 10.1177/0278364918755924.

[BFR07] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. "Programming Self-Organizing Systems with the Higher-Order Chemical Language". In: *Int. J. Unconv. Comput.* 3.3 (2007), pp. 161–177.

[Bil+22] Cem Bilaloglu et al. "A Novel Time-of-Flight Range and Bearing Sensor System for Micro Air Vehicle Swarms". In: *Swarm Intelligence - 13th International Conference, ANTS 2022, Málaga, Spain, November 2-4, 2022, Proceedings*. Ed. by Marco Dorigo et al. Vol. 13491. Lecture Notes in Computer Science. Springer, 2022, pp. 248–256. DOI: 10.1007/978-3-031-20176-9_20. URL: https://doi.org/10.1007/978-3-031-20176-9%5C_20.

[Bla16] S. Blackheath. *Functional Reactive Programming*. Manning, 2016. ISBN: 9781638353416.

[Bon+99] Eric Bonabeau et al. *Swarm intelligence: from natural to artificial systems*. Santa Fe Institute Studies in the Sciences of Complexity. Oxford university press, 1999. ISBN: 978-0-19-513159-8.

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

[Bra+13] Manuele Brambilla et al. "Swarm robotics: a review from the swarm engineering perspective". In: *Swarm Intell.* 7.1 (2013), pp. 1–41. DOI: 10.1007/s11721-012-0075-2. URL: https://doi.org/10.1007/s11721-012-0075-2.

[Cas+19] Roberto Casadei et al. "Aggregate Processes in Field Calculus". In: *Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*. Ed. by Hanne Riis Nielson and Emilio Tuosto. Vol. 11533. Lecture Notes in Computer Science. Springer, 2019, pp. 200–217. DOI: 10.1007/978-3-030-22397-7_12.

[Cas+20a] Roberto Casadei et al. "FScaFi : A Core Calculus for Collective Adaptive Systems Programming". In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12477. Lecture Notes in Computer Science. Springer, 2020, pp. 344–360. DOI: 10.1007/978-3-030-61470-6_21. URL: https://doi.org/10.1007/978-3-030-61470-6%5C_21.

[Cas+20b] Roberto Casadei et al. "Pulverization in Cyber-Physical Systems: Engineering the Self-Organizing Logic Separated from Deployment". In: *Future Internet* 12.11 (2020), p. 203. DOI: 10.3390/fi12110203. URL: https://doi.org/10.3390/fi12110203.

[Cas+21] Roberto Casadei et al. "Engineering collective intelligence at the edge with aggregate processes". In: *Eng. Appl. Artif. Intell.* 97 (2021), p. 104081. DOI: 10.1016/j.engappai.2020.104081. URL: https://doi.org/10.1016/j.engappai.2020.104081.

[Cas+22a] Roberto Casadei et al. "ScaFi: A Scala DSL and Toolkit for Aggregate Programming". In: *SoftwareX* 20 (2022), p. 101248. DOI: 10.1016/j.softx.2022.101248. URL: https://doi.org/10.1016/j.softx.2022.101248.

[Cas+22b] Roberto Casadei et al. "Space-fluid Adaptive Sampling: a Field-based, Self-organising Approach". In: *Coordination Models and Languages - 24th International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*. Ed. by Maurice H ter Beek and Marjan Sirjani. Vol. 13271. Lecture Notes in Computer Science. In press. 2022.

[Cas23] Roberto Casadei. "Macroprogramming: Concepts, State of the Art, and Opportunities of Macroscopic Behaviour Modelling". In: *ACM Comput. Surv.* 55.13s (July 2023). ISSN: 0360-0300. DOI: 10.1145/3579353.

[CJ10] Zheng Chen and Heng Ji. "Graph-Based Clustering for Computational Linguistics: A Survey". In: *Proceedings of the 2010 Workshop on Graph-Based Methods for Natural Language Processing.* TextGraphs-5. Uppsala, Sweden: Association for Computational Linguistics, 2010, pp. 1–9. ISBN: 9781932432770. URL: https://aclanthology.org/W10-2301/.

[CNM17] Nicolás Bulla Cruz, Nadia Nedjah, and Luiza de Macedo Mourelle. "Robust distributed spatial clustering for swarm robotic based systems". In: *Appl. Soft Comput.* 57 (2017), pp. 727–737. DOI: 10.1016/j.asoc.2016.06.002.

[CNS21] Martin Carroll, Kedar S. Namjoshi, and Itai Segall. "The Resh Programming Language for Multirobot Orchestration". In: *IEEE International Conference on Robotics and Automation, ICRA 2021, Xi'an, China, May 30 - June 5, 2021.* IEEE, 2021, pp. 4026–4032. DOI: 10.1109/ICRA48506.2021.9561133. URL: https://doi.org/10.1109/ICRA48506.2021.9561133.

[Coo+06] Ezra Cooper et al. “Links: Web Programming Without Tiers”. In: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*. Ed. by Frank S. de Boer et al. Vol. 4709. Lecture Notes in Computer Science. Springer, 2006, pp. 266–296. DOI: 10.1007/978-3-540-74792-5_12. URL: https://doi.org/10.1007/978-3-540-74792-5%5C_12.

[Cop+20] Mario Coppola et al. “A Survey on Swarming With Micro Air Vehicles: Fundamental Challenges and Constraints”. In: *Frontiers in Robotics and AI* 7 (Feb. 2020). DOI: 10.3389/frobt.2020.00018. URL: https://doi.org/10.3389/frobt.2020.00018.

[CS07] Felipe Cucker and Steve Smale. “Emergent Behavior in Flocks”. In: *IEEE Transactions on Automatic Control* 52.5 (2007), pp. 852–862. DOI: 10.1109/TAC.2007.895842.

[CZ18] Wenyu Cai and Meiyan Zhang. “Spatiotemporal correlation-based adaptive sampling algorithm for clustered wireless sensor networks”. In: *Int. J. Distributed Sens. Networks* 14.8 (2018). DOI: 10.1177/1550147718794614.

[DK18] Dimitris Dedousis and Vana Kalogeraki. “A Framework for Programming a Swarm of UAVs”. In: *11th PErvasive Technologies Related to Assistive Environments Conference (PETRA’18), Proceedings*. ACM, 2018, pp. 5–12. DOI: 10.1145/3197768.3197772. URL: https://doi.org/10.1145/3197768.3197772.

[DM12] Matthew Dunbabin and Lino Marques. “Robots for Environmental Monitoring: Significant Advancements and Applications”. In: *IEEE Robotics Autom. Mag.* 19.1 (2012), pp. 24–39. DOI: 10.1109/MRA.2011.2181683.

[DTT20] Marco Dorigo, Guy Theraulaz, and Vito Trianni. “Reflections on the future of swarm robotics”. In: *Sci. Robotics* 5.49 (2020), p. 4385. DOI: 10.1126/scirobotics.abe4385. URL: https://doi.org/10.1126/scirobotics.abe4385.

[Dun74] Joseph C Dunn. “Well-separated clusters and optimal fuzzy partitions”. In: *Journal of cybernetics* 4.1 (1974), pp. 95–104. DOI: 10.1080/01969727408546059.

[Est02] Vladimir Estivill-Castro. “Why so many clustering algorithms: a position paper”. In: *SIGKDD Explor.* 4.1 (2002), pp. 65–75. DOI: 10.1145/568574.568575.

[Far+17] Alessandro Farinelli et al. “Interacting with team oriented plans in multi-robot systems”. en. In: *Auton. Agent. Multi. Agent. Syst.* 31.2 (Mar. 2017), pp. 332–361.

[GA14] Sahil Garg and Nora Ayanian. “Persistent Monitoring of Stochastic Spatio-temporal Phenomena with a Small Team of Robots”. In: *Robotics: Science and Systems X, University of California, Berkeley, USA, July 12-16, 2014*. Ed. by Dieter Fox, Lydia E. Kavraki, and Hanna Kurniawati. 2014. DOI: 10.15607/RSS.2014.X.038.

[Gel85] David Gelernter. “Generative communication in Linda”. In: *ACM Trans. Program. Lang. Syst.* 7.1 (1985), pp. 80–112. ISSN: 0164-0925. DOI: 10.1145/2363.2433.

[Ger+20] Carlos Gershenson et al. “Self-Organization and Artificial Life”. In: *Artif. Life* 26.3 (2020), pp. 391–408. DOI: 10.1162/artl_a_00324. URL: https://doi.org/10.1162/artl%5C_a%5C_00324.

[Ger07] Carlos Gershenson. *Design and control of self-organizing systems*. CopIt Arxives, 2007.

[GHZ18] Xiaohua Ge, Qing-Long Han, and Xian-Ming Zhang. “Achieving Cluster Formation of Multi-Agent Systems Under Aperiodic Sampling and Communication Delays”. In: *IEEE Trans. Ind. Electron.* 65.4 (2018), pp. 3417–3426. DOI: 10.1109/TIE.2017.2752148.

[Gia17] Kristin Giammarco. “Practical modeling concepts for engineering emergence in systems of systems”. In: *SoSE*. IEEE, 2017, pp. 1–6. DOI: 10.1109/SYSOSE.2017.7994977.

[GLY07] Bugra Gedik, Ling Liu, and Philip S. Yu. “ASAP: An Adaptive Sampling Approach to Data Collection in Sensor Networks”. In: *IEEE Trans. Parallel Distributed Syst.* 18.12 (2007), pp. 1766–1783. DOI: 10.1109/TPDS.2007.1110.

[Gup15] Gopal Gupta. “Language-based software engineering”. In: *Sci. Comput. Program.* 97 (2015), pp. 37–40. DOI: 10.1016/j.scico.2014.02.010. URL: https://doi.org/10.1016/j.scico.2014.02.010.

[Har+20] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[Hos13] Satoshi Hoshino. "Reactive Clustering Method for Platooning Autonomous Mobile Robots". In: *IFAC Proceedings Volumes* 46.10 (2013). 8th IFAC Symposium on Intelligent Autonomous Vehicles, pp. 152–157. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20130626-3-AU-2035.00009. URL: https://www.sciencedirect.com/science/article/pii/S1474667015349259.

[Hu+21] Junyan Hu et al. "A Decentralized Cluster Formation Containment Framework for Multirobot Systems". In: *IEEE Trans. Robotics* 37.6 (2021), pp. 1936–1955. DOI: 10.1109/TRO.2021.3071615.

[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.

[Jai10] Anil K. Jain. "Data clustering: 50 years beyond K-means". In: *Pattern Recognition Letters* 31.8 (2010), pp. 651–666. ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2009.09.011.

[Jav+18] Muhammad Aqib Javed et al. "Community detection in networks: A multidisciplinary review". In: *J. Netw. Comput. Appl.* 108 (2018), pp. 87–111. DOI: 10.1016/j.jnca.2018.02.011.

[JMF99] A. K. Jain, M. N. Murty, and P. J. Flynn. "Data Clustering: A Review". In: *ACM Comput. Surv.* 31.3 (1999), pp. 264–323. ISSN: 0360-0300. DOI: 10.1145/331499.331504.

[KB91] Yoram Koren and Johann Borenstein. "Potential field methods and their inherent limitations for mobile robot navigation". In: *Proceedings of the 1991 IEEE International Conference on Robotics and Automation, Sacramento, CA, USA, 9-11 April 1991*. IEEE Computer Society, 1991, pp. 1398–1404. DOI: 10.1109/ROBOT.1991.131810. URL: https://doi.org/10.1109/ROBOT.1991.131810.

[KC03] Jeffrey O. Kephart and David M. Chess. "The Vision of Autonomic Computing". In: *Computer* 36.1 (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055. URL: https://doi.org/10.1109/MC.2003.1160055.

[Kem+17] Stephanie Kemna et al. "Multi-robot coordination through dynamic Voronoi partitioning for informative adaptive sampling in communication-constrained environments". In: *2017 IEEE International Conference on Robotics and Automation, ICRA 2017, Singapore, Singapore, May 29 - June 3, 2017*. IEEE, 2017, pp. 2124–2130. DOI: 10.1109/ICRA.2017.7989245.

[KGB21] Miquel Kegeleirs, Giorgio Grisetti, and Mauro Birattari. "Swarm SLAM: Challenges and Perspectives". In: *Frontiers in Robotics and AI* 8 (Mar. 2021). DOI: 10.3389/frobt.2021.618268. URL: https://doi.org/10.3389/frobt.2021.618268.

[KH04] Nathan P. Koenig and Andrew Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, Sendai, Japan, September 28 - October 2, 2004*. IEEE, 2004, pp. 2149–2154. DOI: 10.1109/IROS.2004.1389727.

[KL16] Manos Koutsoubelias and Spyros Lalis. "TeCoLa: A Programming Framework for Dynamic and Heterogeneous Robotic Teams". In: *Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2016)*. ACM, 2016, pp. 115–124. DOI: 10.1145/2994374.2994397. URL: https://doi.org/10.1145/2994374.2994397.

[Kos+20] Oliver Kosak et al. "Maple-Swarm: Programming Collective Behavior for Ensembles by Extending HTN-Planning". In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12477. Lecture Notes in Computer Science. Springer, 2020, pp. 507–524. DOI: 10.1007/978-3-030-61470-6_30. URL: https://doi.org/10.1007/978-3-030-61470-6%5C_30.

[Kuc+20] Kerem Kucuk et al. "Crowd sensing aware disaster framework design with IoT technologies". In: *J. Ambient Intell. Humaniz. Comput.* 11.4 (2020), pp. 1709–1725. DOI: 10.1007/s12652-019-01384-1.

[Lim+18] Keila Lima et al. "Dolphin: A Task Orchestration Language for Autonomous Vehicle Networks". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2018, Madrid, Spain, October 1-5, 2018*. IEEE, 2018, pp. 603–610. DOI: 10.1109/IROS.2018.8594059. URL: https://doi.org/10.1109/IROS.2018.8594059.

[LKK05] C. Lee, M. Kim, and Sanza Kazadi. "Robot Clustering". In: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Waikoloa, Hawaii, USA, October 10-12, 2005*. IEEE, 2005, pp. 1449–1454. DOI: 10.1109/ICSMC.2005.1571350.

[LM07] Yen-Ting Lin and Seapahn Megerian. “Sensing Driven Clustering for Monitoring and Control Applications”. In: *4th IEEE Consumer Communications and Networking Conference, CCNC 2007, Las Vegas, NV, USA, January 11-13, 2007*. IEEE, 2007, pp. 202–206. DOI: 10.1109/CCNC.2007.47.

[Luc+19] Matt Luckcuck et al. “Formal Specification and Verification of Autonomous Robotic Systems: A Survey”. In: *ACM Comput. Surv.* 52.5 (2019), 100:1–100:41. DOI: 10.1145/3342355. URL: https://doi.org/10.1145/3342355.

[MBD18] Yuanqiu Mo, Jacob Beal, and Soura Dasgupta. “An Aggregate Computing Approach to Self-Stabilizing Leader Election”. In: *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Trento, Italy, September 3-7, 2018*. IEEE, 2018, pp. 112–117. DOI: 10.1109/FAS-W.2018.00034. URL: https://doi.org/10.1109/FAS-W.2018.00034.

[MBF11] D. Martens, B. Baesens, and T. Fawcett. “Editorial survey: swarm intelligence for data mining”. In: *Machine Learning* 82 (2011), pp. 1–42. DOI: https://doi.org/10.1007/s10994-010-5216-5.

[Mek+18] Kais Mekki et al. “Overview of Cellular LPWAN Technologies for IoT Deployment: Sigfox, LoRaWAN, and NB-IoT”. In: *2018 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2018, Athens, Greece, March 19-23, 2018*. IEEE Computer Society, 2018, pp. 197–202. DOI: 10.1109/PERCOMW.2018.8480255. URL: https://doi.org/10.1109/PERCOMW.2018.8480255.

[MH12] Georg Martius and J. Michael Herrmann. “Variants of guided self-organization for robot control”. In: *Theory Biosci.* 131.3 (2012), pp. 129–137. DOI: 10.1007/s12064-011-0141-0.

[Mot+14] Luca Mottola et al. “Team-level programming of drone sensor networks”. In: *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys’14)*. ACM, 2014, pp. 177–190. DOI: 10.1145/2668332.2668353. URL: https://doi.org/10.1145/2668332.2668353.

[MP11] Luca Mottola and Gian Pietro Picco. “Programming wireless sensor networks: Fundamental concepts and state of the art”. In: *ACM Comput. Surv.* 43.3 (2011), 19:1–19:51. DOI: 10.1145/1922649.1922656. URL: https://doi.org/10.1145/1922649.1922656.

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

[NJW20] Rocco De Nicola, Stefan Jähnichen, and Martin Wirsing. “Rigorous engineering of collective adaptive systems: special section”. In: *Int. J. Softw. Tools Technol. Transf.* 22.4 (2020), pp. 389–397. DOI: 10.1007/s10009-020-00565-0.

[NMW07] Ryan Newton, Greg Morrisett, and Matt Welsh. “The regiment macroprogramming system”. In: *Proceedings of the 6th International Conference on Information Processing in Sensor Networks, IPSN 2007.* ACM, 2007, pp. 489–498. DOI: 10.1145/1236360.1236422. URL: https://doi.org/10.1145/1236360.1236422.

[Noo+19] Joseph Noor et al. “DDFlow: visualized declarative programming for heterogeneous IoT networks”. In: *IoTDI.* ACM, 2019, pp. 172–177. DOI: 10.1145/3302505.3310079.

[OD01] Andrea Omicini and Enrico Denti. “From tuple spaces to tuple centres”. In: *Sci. Comput. Program.* 41.3 (2001), pp. 277–294. DOI: 10.1016/S0167-6423(01)00011-9.

[PB15] H. Van Dyke Parunak and Sven A. Brueckner. “Software engineering for self-organizing systems”. In: *Knowl. Eng. Rev.* 30.4 (2015), pp. 419–434. DOI: 10.1017/S0269888915000089.

[PB16a] Carlo Pinciroli and Giovanni Beltrame. “Buzz: A Programming Language for Robot Swarms”. In: *IEEE Softw.* 33.4 (2016), pp. 97–100. DOI: 10.1109/MS.2016.95. URL: https://doi.org/10.1109/MS.2016.95.

[PB16b] 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 2016, Daejeon, South Korea, October 9-14, 2016.* IEEE, 2016, pp. 3794–3800. DOI: 10.1109/IROS.2016.7759558. URL: https://doi.org/10.1109/IROS.2016.7759558.

[Pen+15] Yan Peng et al. “The obstacle detection and obstacle avoidance algorithm based on 2-D lidar”. In: *IEEE International Conference on Information and Automation, ICIA 2015, Lijiang, China, August 8-10, 2015.* IEEE, 2015, pp. 1648–1653. DOI: 10.1109/ICInfA.2015.7279550. URL: https://doi.org/10.1109/ICInfA.2015.7279550.

[Pha+10] Ngoc Duy Pham et al. "SCCS: Spatiotemporal clustering and compressing schemes for efficient data collection applications in WSNs". In: *Int. J. Commun. Syst.* 23.11 (2010), pp. 1311–1333. DOI: 10.1002/dac.1104.

[Pia+21a] Danilo Pianini et al. "Partitioned integration and coordination via the self-organising coordination regions pattern". In: *Future Gener. Comput. Syst.* 114 (2021), pp. 44–68. DOI: 10.1016/j.future.2020.07.032.

[Pia+21b] Danilo Pianini et al. "Time-Fluid Field-Based Coordination through Programmable Distributed Schedulers". In: *Log. Methods Comput. Sci.* 17.4 (2021). URL: https://doi.org/10.46298/lmcs-17(4:13)2021.

[PMV13] D Pianini, S Montagna, and M Viroli. "Chemical-oriented simulation of computational systems with ALCHEMIST". In: *Journal of Simulation* 7.3 (Aug. 2013), pp. 202–215. DOI: 10.1057/jos.2012.27. URL: https://doi.org/10.1057/jos.2012.27.

[Pro09] Mikhail Prokopenko. *Guided self-organization.* 2009.

[Rey87] Craig W. Reynolds. "Flocks, herds and schools: A distributed behavioral model". In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987, Anaheim, California, USA, July 27-31, 1987.* Ed. by Maureen C. Stone. ACM, 1987, pp. 25–34. DOI: 10.1145/37401.37406. URL: https://doi.org/10.1145/37401.37406.

[Rou87] Peter J. Rousseeuw. "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis". In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65. ISSN: 0377-0427. DOI: https://doi.org/10.1016/0377-0427(87)90125-7. URL: https://www.sciencedirect.com/science/article/pii/0377042787901257.

[RTG19] Nathalie Barbosa Roa, Louise Travé-Massuyès, and Victor Hugo Grisales. "*DyClee*: Dynamic clustering for tracking evolving environments". In: *Pattern Recognit.* 94 (2019), pp. 162–186. DOI: 10.1016/j.patcog.2019.05.024.

[Sae+10] Joan Saez-Pons et al. "Multi-robot team formation control in the GUARDIANS project". en. In: *Ind. Rob.* 37.4 (June 2010), pp. 372–383.

[Say09] Hiroki Sayama. "Swarm Chemistry". In: *Artif. Life* 15.1 (2009), pp. 105–114. DOI: 10.1162/artl.2009.15.1.15107.

[Sch+10] Hartmut Schmeck et al. "Adaptivity and self-organization in organic computing systems". In: *ACM Trans. Auton. Adapt. Syst.* 5.3 (2010), 10:1–10:32. DOI: 10.1145/1837909.1837911.

[Sch+20] Melanie Schranz et al. "Swarm Robotic Behaviors and Current Applications". In: *Frontiers Robotics AI* 7 (2020). DOI: 10.3389/frobt.2020.00036.

[Sen+22] Iwens Gervásio Sene Júnior et al. "The state of the art of macroprogramming in IoT: An update". In: *J. Internet Serv. Appl.* 13.1 (2022), pp. 54–65. DOI: 10.5753/jisa.2022.2372. URL: https://doi.org/10.5753/jisa.2022.2372.

[SSP13] Vivek Kumar Singh, Garima Singh, and Suparna Pande. "Emergence, Self-Organization and Collective Intelligence - Modeling the Dynamics of Complex Collectives in Social and Organizational Settings". In: *UKSim*. IEEE, 2013, pp. 182–189. DOI: 10.1109/UKSim.2013.77.

[SW22] Pradeep Sambu and Myounggyu Won. "An Experimental Study on Direction Finding of Bluetooth 5.1: Indoor vs Outdoor". In: *IEEE Wireless Communications and Networking Conference, WCNC 2022, Austin, TX, USA, April 10-13, 2022*. IEEE, 2022, pp. 1934–1939. DOI: 10.1109/WCNC51071.2022.9771930. URL: https://doi.org/10.1109/WCNC51071.2022.9771930.

[Tes+22] Lorenzo Testa et al. "Aggregate Processes as Distributed Adaptive Services for the Industrial Internet of Things". In: *Pervasive and Mobile Computing* (to appear) (2022).

[TM03] Robert Tolksdorf and Ronaldo Menezes. "Using Swarm Intelligence in Linda Systems". In: *Engineering Societies in the Agents World IV, 4th International Workshop, ESAW 2003, London, UK, October 29-31, 2003, Revised Selected and Invited Papers*. Ed. by Andrea Omicini, Paolo Petta, and Jeremy Pitt. Vol. 3071. Lecture Notes in Computer Science. Springer, 2003, pp. 49–65. DOI: 10.1007/978-3-540-25946-6_3.

[TU21] Michael C. Thrun and Alfred Ultsch. "Swarm intelligence for self-organized clustering". In: *Artificial Intelligence* 290 (2021). ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2020.103237.

[Var+15] Franck Varenne et al. "Programming the emergence in morphogenetically architected complex systems". In: *Acta biotheoretica* 63.3 (2015), pp. 295–308. DOI: 10.1007/s10441-015-9262-z.

[Vic+95] Tamás Vicsek et al. "Novel Type of Phase Transition in a System of Self-Driven Particles". In: *Phys. Rev. Lett.* 75 (6 Aug. 1995), pp. 1226–1229. DOI: 10.1103/PhysRevLett.75.1226. URL: https://link.aps.org/doi/10.1103/PhysRevLett.75.1226.

[Vir+18] Mirko Viroli et al. "Engineering Resilient Collective Adaptive Systems by Self-Stabilisation". In: *ACM Trans. Model. Comput. Simul.* 28.2 (2018), 16:1–16:28. DOI: 10.1145/3177774. URL: https://doi.org/10.1145/3177774.

[Vir+19] Mirko Viroli et al. "From distributed coordination to field calculus and aggregate computing". In: *J. Log. Algebraic Methods Program.* 109 (2019). DOI: 10.1016/j.jlamp.2019.100486. URL: https://doi.org/10.1016/j.jlamp.2019.100486.

[WH00] Zhanyong Wan and Paul Hudak. "Functional reactive programming from first principles". In: *PLDI*. ACM, 2000, pp. 242–252. DOI: 10.1145/349299.349331.

[WKS18] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. "Distributed system development with ScalaLoci". In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 129:1–129:30. DOI: 10.1145/3276499. URL: https://doi.org/10.1145/3276499.

[WS19] Pascal Weisenburger and Guido Salvaneschi. "Multitier Modules". In: *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 3:1–3:29. DOI: 10.4230/LIPIcs.ECOOP.2019.3. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2019.3.

[WS20] Pascal Weisenburger and Guido Salvaneschi. "Implementing a Language for Distributed Systems: Choices and Experiences with Type Level and Macro Programming in Scala". In: *Art Sci. Eng. Program.* 4.3 (2020), p. 17. DOI: 10.22152/programming-journal.org/2020/4/17. URL: https://doi.org/10.22152/programming-journal.org/2020/4/17.

[WWS20] Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. "A Survey of Multitier Programming". In: *ACM Comput. Surv.* 53.4 (2020), 81:1–81:35. DOI: 10.1145/3397495. URL: https://doi.org/10.1145/3397495.

[Wyc+98] P. Wyckoff et al. "T Spaces". In: *IBM Systems Journal* 37.3 (1998), pp. 454–474. DOI: 10.1147/sj.373.0454.

[Yi+20] Wei Yi et al. “An Actor-based Programming Framework for Swarm Robotic Systems”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021*. IEEE, 2020, pp. 8012–8019. DOI: 10.1109/IROS45743.2020.9341198. URL: https://doi.org/10.1109/IROS45743.2020.9341198.

Part III

Learning in Cyber-Physical Swarms

Chapter 10

Research Roadmap for Hybrid aggregate Computing

What is the research roadmap toward hybrid aggregate computing?
What are the opportunities and challenges of this research?
How can we address these challenges?
– **RQ1, RQ2**

Contents

10.1 Roadmap 204
10.1.1 Goals and Means 204
10.1.2 Patterns: learning AC algorithms 205
10.1.3 Platform: learning execution strategies and adaptations 206
10.1.4 Platform: learning system structures and re-structuring 207
10.2 Opportunities and Challenges 208
10.3 Final Remarks 209

Engineering of AC applications is a rich activity that spans multiple concerns including designing the aggregate program, developing reusable algorithms [Aud+17; Aud+21], detailing the execution model [Pia+21], and choosing a deployment based on available infrastructure [Cas+20b] (see Figure 10.1 for the full AC "stack"). Traditionally, these activities have been carried out through ad-hoc designs and implementations created by developers and tailored to specific contexts and goals (see Part II of the thesis), leading to, e.g., self-organization algorithms that are very reactive under certain network assumptions [Aud+21], or

round execution frequencies tailored to the velocity of change of underlying environment phenomena [Pia+21]. To overcome the complexity and cost of manually tailoring or devising general but inefficient algorithms, execution details, and deployments, we propose to use *machine learning (ML) techniques*. In particular, we observe that automated design driven by learning can be applied at different levels of the AC "engineering stack". The integration of AC and ML done during this thesis, which we refer to as aggregate computing + machine learning (AC+ML) or *hybrid* aggregate computing, is provided a lot of opportunities and challenges, fostering a long-term record of research contributions. In this chapter, we discuss a research roadmap followed during the thesis, for achieving the vision of hybrid aggregate computing.

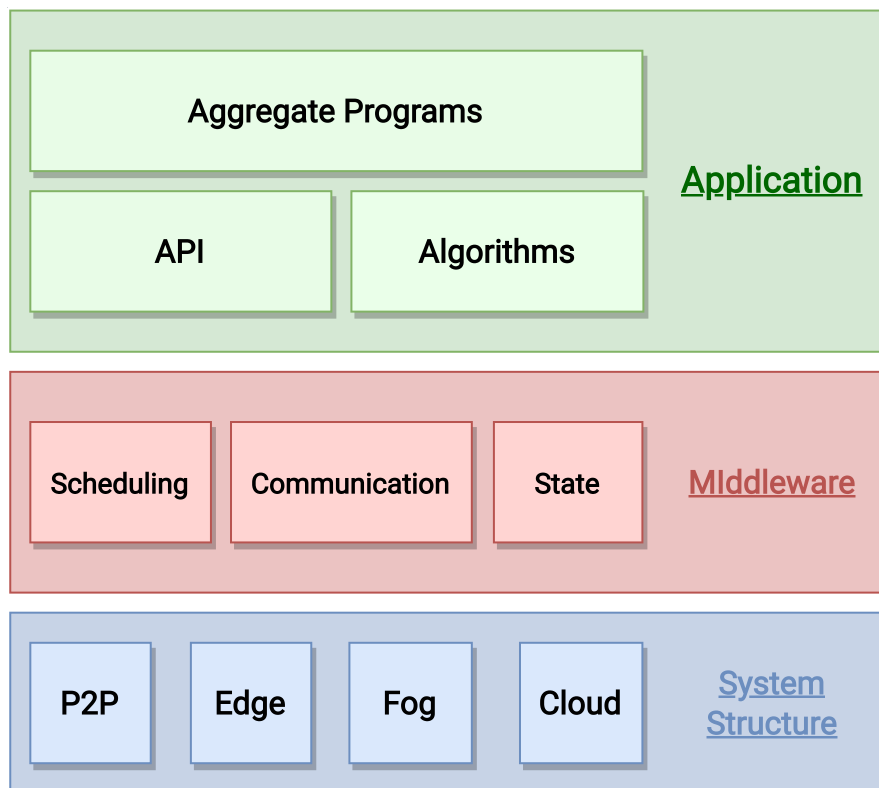


Figure 10.1: The entire Aggregate Computing stack. The application level (composed of API, algorithms and programs) needs to be supported by an execution platform/middleware, to decouple the application from systems structures. Then, this execution platform should be deployed in a particular architecture. In our vision, machine learning could enhance all of these layers.

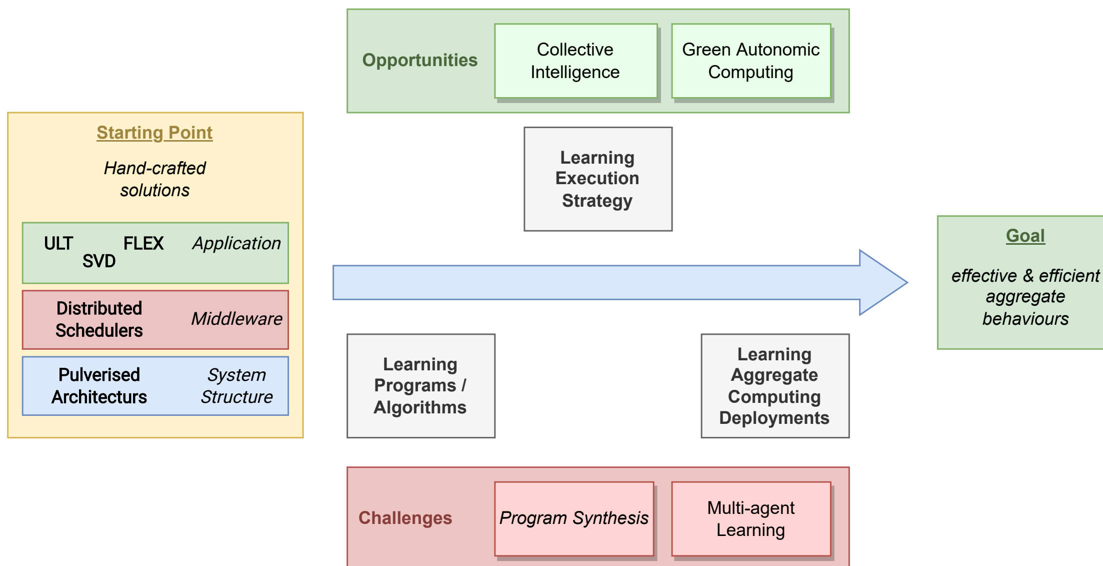


Figure 10.2: Overview of the research roadmap aiming at efficient and effective aggregate computations. The starting point is current research, based on hand-crafted solutions. The goal is addressed through the application of hybrid aggregate computing at the program/execution/deployment levels.

10.1 Roadmap

This section details motivation and direction for AC+ML research, also summarized in Figure 10.2.

10.1.1 Goals and Means

To systematically analyse the ways in which ML can promote the development of AC applications, it is important to consider the *goals* and *means* of the AC framework.

The goals include:

1. *functionality* — achieving some collective behaviour (e.g., environment monitoring and control through sensor-actuator networks [Cas+20b; Pia+21], resiliently organize the system into leader-regulated areas [Cas+19], and matching and coordinating collective tasks with worker ensembles [Cas+21]);

2. *non-functionality* — concerning with the cost associated to the functionality.

In particular, the latter can be further divided into multiple sub-goals from which application-specific *trade-offs* can be made:

1. *time efficiency*: refers to the time needed to converge to the desired state-of-affairs;
2. *communication efficiency*: refers to the amount of communication performed (e.g., measured in terms of messages or bandwidth);
3. *execution efficiency*: refers to the number of rounds of computations performed;
4. *energy efficiency*: e.g. by combining communication and execution efficiency;
5. *dependability*: concerns e.g. reliability or safety of a collective and its products.

In the AC approach, these goals can be addressed through three main means:

1. algorithms;
2. execution strategy;
3. system structure (deployment).

Now, it turns out that ML could be a powerful technique to replace or augment those traditionally human-engineered means.

10.1.2 Patterns: learning AC algorithms

AC algorithms take collective inputs and use computational mechanisms to produce collective outputs. In the *field calculus* [Vir+19], the minimal formal framework that underpins AC, collective inputs and outputs are denoted by *computational fields* (or *fields* for short), namely distributed data structures mapping each device of the aggregate system to a value. The field calculus, then, provides a set of operators for manipulating fields: essentially, operators specifying how fields evolve over time (round after round), and operators specifying interactions with neighbours (which can be reified through *neighbouring fields*). So, in AC, a collective adaptive behaviour is the result of an algorithm (a function from fields to fields) expressed e.g. in ScaFi and the concrete execution of the algorithm in a system of devices.

Usually, algorithms are *progressive* – they take time (computation rounds and communications) to converge to the "correct" value – and *self-healing*, i.e., they can adjust their output following changes in their inputs and the system topology. For instance, a gradient algorithm progressively corrects the field of distances after the set of sources changes, or nodes move (hence changing the distances between neighbour nodes) or enter/leave the system.

So, different gradient algorithms may achieve the *same* functionality (i.e., the eventual computation of the field of minimum distances from sources) with different non-functional outcomes. Indeed, they may:

- take a different time or a different number of rounds to converge, for the same initial condition;
- require a different amount of data to be exchanged;
- take different trade-offs e.g. regarding reactivity and smoothness (cf. the stability of values during change) [Aud+21; Aud+17]; or
- take different assumptions regarding the execution model or the environment [Aud+21], which may affect applicability.

Designing efficient and versatile AC algorithms can be complex [Aud+21; Aud+17]: therefore, it is interesting to explore whether algorithms can be learnt or synthesized given high-level functional goals. For this purpose, an idea could be to combine the program synthesis [GPS17] technique called *sketching* [Sol08] with machine learning. With this approach, the designer could provide an algorithm template with holes corresponding to actions to be learnt by the agents or the whole collective, e.g., through reinforcement learning. In this case, learning would be used to *search* for an optimal policy. The resulting algorithm, then, would need extensive testing (e.g. by simulation) in a representative set of environments and dynamics. In this regard, research is needed to identify what synthesis techniques, learning algorithms, frameworks, and methodologies can support the learning of algorithms able to achieve performance similar or better than state-of-the-art solutions. The first efforts in this direction are highlighted in Chapters 11 and 12.

10.1.3 Platform: learning execution strategies and adaptations

For a given AC program or algorithm, multiple execution strategies can be applied, affecting aspects like the scheduling of computation rounds (i.e., the frequency of execution), the scheduling of communications (i.e., when the devices exchange data), the retention of messages from neighbours (i.e., when should messages from the neighbour be considered too old to be used). In particular, a first distinction can be made between *static* and *dynamic* execution strategies. The latter approach adapts the execution choices at runtime depending on factors which may include the speed of environmental change, the energy level of a device, incentives in volunteering settings, or the desired Quality of Service (QoS). Moreover, these factors may be diverse in diverse portions of the system; so, it is in general important to also consider the local context of each device or set of devices.

Note that adaptive behaviour could be achieved via static execution strategies, e.g., by using reactive approaches triggering behaviours when specific context conditions apply [Pia+21]. However, again, since it is in general hard to design static or dynamic execution strategies able to adequately take into account all the factors and goals, it could make sense to let a system (and its components) learn how to efficiently execute algorithms according to a set of given high-level objectives. Indeed, true adaptiveness comes from changing the behavioural rules, and learning is a premier tool for changing for the better. The emphasis on improving efficiency by optimizing execution of aggregate systems, hence promoting sustainability of collective computations, could be the opportunity to open up a vision of *green autonomic computing*. In Chapter 13 we propose a reinforcement learning approach to learn the execution strategy to improve the execution efficiency of a gradient algorithm.

10.1.4 Platform: learning system structures and restructuring

A logical AC system consists of a logical network of logical devices operating as per the aforementioned execution protocol. It is the *collective digital twin* [Cas+22a] of a target set of application-level physical devices (e.g., robots of a swarm, or workers in a computing ecosystem). As shown in recent work on *pulverized architectures* [Cas+20b], it turns out that different application partitioning schemas and implementations of the *digital thread* associated with the aggregate system are possible, as well as different deployments of application components onto the available Information-Communication Technology (ICT) infrastructure. For instance, it is possible to embed evaluations of the aggregate program into the devices themselves, to move the entire computational part to the cloud (leaving devices as thin hosts dealing only with sensing and actuation), or spread these onto a layer of edge-fog infrastructural devices. Different deployments may lead to different efficiency trade-offs and non-functional outcomes [Cas+20b; Cas+22a], which, crucially, may also change dynamically due to application and infrastructure dynamics (cf. addition or removal of new nodes, blackouts, etc.).

In previous research, aggregate application partitioning and deployment have been done manually at design time [Cas+20b]—as showed in Chapter 9. However, for a given set of infrastructures, it is not easy to determine an *effective* mapping. The usual approach consists of manually generating different deployments, simulating these for the same set of applications, collecting various cost metrics, and evaluating results to determine trade-offs and guidelines. However, ML could be *injected* into such a methodology to have the system learn by itself what is a (locally) optimal deployment for an aggregate application. Moreover, the system

could be induced to learn a strategy to self-adapt the deployment (i.e., by moving components opportunistically across the ICT infrastructure) while trying to preserve, e.g., certain QoS targets.

Additionally, it has been shown in [Cas+22a], that changing the logical structure of an aggregate system at runtime (e.g., by injecting *virtual* devices) could be a further means for steering self-organization processes, namely to improve the collective behaviour of a system (cf. [LS11]). In this respect, a challenge would be to determine *how*, *when*, and *where* virtual devices should be spawned or removed from the system. In the AC+ML vision, this problem should not be addressed through ad-hoc solutions, but the AC system should be trained in order to learn the best strategies for improving the efficacy and efficiency of aggregate applications.

10.2 Opportunities and Challenges

Opportunities

A prominent opportunity of AC+ML research lies in potentially getting insights about the *automatic design of Collective Intelligence (CI)*, renewing the partial contributions given by Szuba's computational collective intelligence [Szu01] and Tumer and Wolpert's COIN (COllective INtelligence) [WT02]. However, unlike previous work, the peculiar characteristic of aggregate computing of reifying CI into macro-level *programs* (which we may refer to as *CI programmability*) is expected to enable a synergic and gentle introduction of automatic design and learning. Additionally, the other crucial aspect of functional *compositionality* of aggregate behaviours (denoted by functions operating on fields), is also expected to help, e.g., by fostering learning processes whereby the goal is to find suitable compositions of elementary collective behaviours. Moreover, the ability to change execution strategies while guaranteeing the same behaviour could also be considered a form of CI.

Indeed, another key opportunity lies in the possibility of fostering *efficiency* in large-scale intelligent systems, which is more and more important for sustainability as advocated by important fields like *green computing* [SM16]. The significance of the problem is especially relevant nowadays because of the tension between the visions of pervasive computing [SM03], future-generation large-scale computational collectives [Cas+21], and autonomic computing (promoting smarter – i.e. more computationally intensive – devices) [KC03] and the urge to limit the impact of humans and technologies on the environment.

Technical Challenges

Applying learning through the AC stack (Figure 10.1) poses several challenges, many of which are implied by the nature of aggregate systems—cf. distribution, decentralization, partial observability, many-agents coordination, and the eventual nature of collective computations,

Particularly, learning in *many*-agent networked system [ZYB21] is currently an open challenge. Indeed, extending the learning from one agent to many agents exponentially enlarges the policy search space, due to the combinatorial nature of multi-agent systems [HKT19]. Moreover, the neighbouring-based system structure is not strict and could evolve in time, leading to time- and space-varying input spaces. Furthermore, it is not appropriate to use a centralized controller that orchestrates the system as a whole, due to the typical large scale and resilience requirements. However, using only a local vision of the system could lead to the problem of *non-stationarity*, since each agent concurrently learns and modifies the environment in the eye of the other agents [TW12]. Another concern related to many-agent settings is the *multi-agent credit assignment* problem [SB18]. This is referred to in the difficulty of deriving a local reward policy from a global utility that measures the system as a whole [AT04]. Besides, the reward received is typically very delayed and sparse in time, because an action taken in a point of large geographical space, could lead to an influence on the whole system only when it reaches all nodes.

These challenges are mitigated by an increasing track of research records on AC [Vir+19], the availability of formal tools for analysing and reasoning about aggregate computations and systems [Vir+19], and the support given by tools for developing and simulating aggregate systems [Vir+19].

10.3 Final Remarks

Developing AC applications requires addressing various algorithmic, computational, and deployment concerns. The integration of ML across the AC development stack provides opportunities and challenges requiring significant research. The roadmap for AC+ML delineated in this thesis is expected to provide results and insights on the engineering of collectively intelligent distributed systems. In the ensuing chapters, we will elaborate on the efforts delineated by this roadmap, which align with the facets defined during the engineering phase. Specifically concerning high-level aspects (i.e., the design patterns), we first introduce an approach called "Collective Program Sketching" Chapter 10. This method enables the definition of partial collective algorithms with intentional gaps, subsequently filled via machine learning techniques. Additionally, we introduce a distinct yet

related concept called “Field-informed reinforcement learning” Chapter 12 . In this context, aggregate computing is employed to enhance the collective learning process. The computational fields generated by aggregate computing serve to inform the learning process with collective knowledge, despite the existence of only local knowledge.

Subsequently, we shifted our focus to platform facets. The initial endeavour in this realm culminated in the development of “distributed schedulers” chapter 13, wherein RL has been utilized to augment the efficiency of a self-stabilizing collective computation. The experience gained from these three projects has led to the creation of a specialized tool named “ScaRLib” Chapter 14. This tool serves as an enabling medium towards a hybrid vision of aggregate computing and is particularly well-suited for supporting many-agent learning vision.

Chapter 11

Patterns: Collective Program Sketching

Contents

11.1 Aggregate Programs Improvement through RL 213
11.2 Motivation: Building blocks Refinement 213
11.3 Learning Schema 214
11.4 Reinforcement learning-based gradient block 216
11.5 Evaluation 217
11.5.1 Simulation setup 218
11.5.2 Results and Discussion 219
11.6 Final Remarks 221

CPSW behaviour can be expressed by a single *aggregate program* (global perspective) that also defines what processing and communication activities must be performed by each individual device (local perspective).

Besides the programming model and its implications, a significant portion of research on AC [Vir+19] has focussed on design and analysis of *coordination algorithms* expressed in field calculus (FC) for efficiently carrying out self-organising behaviours like, e.g., computing fields of minimum distances from sources (*gradients*) [NSB03; MZL04; Aud+17], electing leaders [MBD18], or distributed summarization [Aud+21]. However, devising self-organising coordination algorithms is not easy; especially difficult is identifying solutions that are efficient across environment assumptions, configurations, and perturbations. The difficulty lies in determining, for a current context, the local decisions of each device, in terms e.g. of processing steps and communication acts, producing output fields that quickly converge to the correct solution.

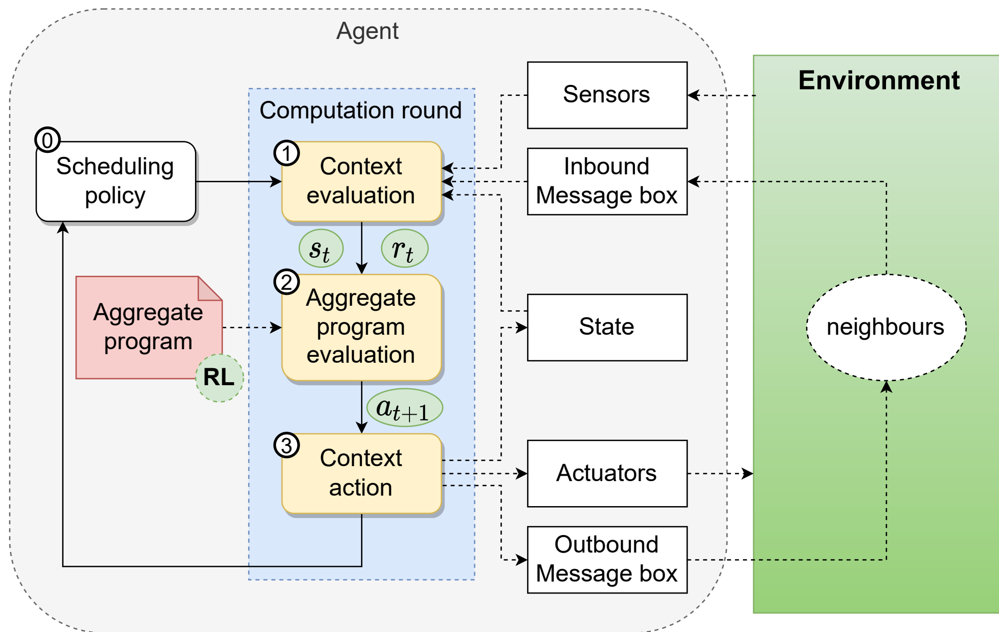


Figure 11.1: Integration of RL within the AC control architecture [CAV21]. The RL state and reward concepts build upon the context, given by environment and neighbour data. The designer configures action points where learning can improve the aggregate computation. The actions selected by the learned policies will then affect the environment (via actuators) and neighbours (via outbound messages).

In chapter we adopt a *RL-based approach*—where an agent learns from experience how to behave in order to maximize delayed cumulative rewards. We devise a general methodology that somewhat resembles the notion of *sketching* in program synthesis [Sol08]: a template program is given and *holes* are filled with actions determined through search. In our case, the program is the AC specification of a coordination algorithm, and holes are filled with actions of a policy learnt through many-agent Hysteretic Q-Learning. We consider the case of the classic gradient algorithm, a paradigmatic and key building block of self-organising coordination [Vir+19; Bea+13; Cas22]: we show via simulations that the system, after sufficient training, learns an improved way to compute and adjust gradient fields to network perturbations.

11.1 Aggregate Programs Improvement through RL

As anticipated in Chapter 3, the behaviour of an aggregate system depends on the interplay of two main ingredients:

- the aggregate program, expressing conceptually the global behaviour of the entire system, and concretely the local behaviour of each node in terms of local processing and data exchange with neighbours; and
- the aggregate execution model, promoting certain dynamics of the system in terms of topology management (e.g., via neighbour discovery), execution of rounds, scheduling of communications; and

While the latter cannot be controlled, the importance of the first element is reflected in the research on the design of novel algorithms (cf. [Vir+19; Aud+17]), while the second element is studied w.r.t. the possibility of tuning and adaptivity according to available technology and infrastructure or the dynamics of the environment—see Chapter 13. Since tuning programs or execution details to specific environments or adapting those to changing environments can be burdensome, it makes sense to consider the use of machine-learning techniques to let a system *learn* effective strategies for unfolding collective adaptive behaviour. In this chapter, we focus on *improving aggregate programs* by learning effective local actions within a given algorithmic schema—an approach similar to the *sketching* technique for program synthesis [Sol08]. Differently from the previous chapter, where we used RL to improve the execution model, here we use RL to improve the aggregate program. Specifically, we integrate RL within the AC control architecture to support the learning of good collective behaviour sketched by a given aggregate program (Figure 11.1): we focus on improving AC building blocks (such as the gradient algorithm covered in Part I) through learning, leading toward a so-called *reinforcement learning-based aggregate computing*. Learning, thus, does not replace the AC methodology for defining the programs, but it is best understood as a technique that supports and improves the AC algorithm design process.

11.2 Motivation: Building blocks Refinement

A major advantage of AC as a programming model is its *compositionality*: complex collective behaviours (e.g., the maintenance of a multi-hop connectivity channel) can be expressed through a combination of building blocks capturing simpler collective behaviours (e.g., gradients). Since building blocks are fundamental bricks of

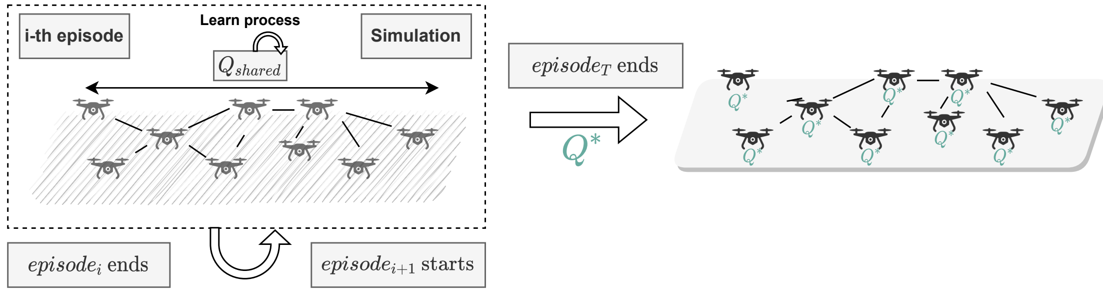


Figure 11.2: Reinforcement Learning schema used in our simulations. The learning algorithm is applied at simulation time (for T episodes) improving a shared Q table. At the deployment time then, the agents exploit a local copy of the optimal Q^* table found by learning.

behaviour that often recur in programs, their bad and good qualities (cf., convergence speed, stability, etc.) tend to amplify and affect behaviours that depend on them. Therefore, research tends to investigate refined variants of building blocks that provide the same functionality but are more *effective* or *efficient* under specific assumptions or contexts (e.g., high mobility, situations where stability is more important than precision, etc.) [Aud+17; Aud+21]. With a library of algorithms, the designer can choose the best combination of building blocks that are well-fitted for a given environment, and even substitute a building block with a variant implementation without substantially affecting the application logic. In general, a building block can be seen as a *black box* (function) that takes a set of input fields (e.g. metric, perception fields, constant fields, etc.) and yields an output field. To increase its flexibility, such a function could leverage a *refinement policy* able to affect the behaviour of the building block over time or in a certain situation. This policy could be a feedback loop, hysteresis, or custom logic to solve a specific problem. We aim at structuring the learning of refinement policies through RL [Agu21]. Our idea is that it should not be the designer who codes a particular block to be used, but that it is the learning algorithm that understands, given a global policy to be optimized following a utility function, what actions need to be activated.

11.3 Learning Schema

The learning algorithm is seen as a state (s_t) evolution function in which the nodes try to apply a correction factor (`update`) following a policy (π^Q_{target} or $\pi^Q_{behavioral}$) refined by learning. The state is built from a neighbourhood field of the building block generic output (o_t) passed as input. Listing 11.1 exemplifies the general

```
def optBlock(o_{t-1}) { // learning as a field that evolves in time
  rep((s_0,a_0, o_0)) { // s_0, a_0 context dependent
    case (s_{t-1},a_{t-1}, _) => {
      val Q = sense("Q") // global during training, local during execution
      val o_t = update(o_{t-1}, a_{t-1}) // local action
      // state from the neighbourhood field program output
      val s_t = state(nbr(o_t))
      val a_t = branch(learn) { // actions depends on learn condition
        val r_{t-1} = reward(o_t, simulation) // simulation is a global object
        simulation.updateQ(Q, s_{t-1}, a_{t-1}, r_{t-1}, s_t) // Q update
        ~ π^Q_{behavioural}(s_t) // sample from a probabilistic distribution
      } {
        π^Q_{target}(s_t) // greedy policy, no sampling is needed
      }
    }
    (s_t, a_t, o_t)
  }._3 // select the output from the tuple
}
```

Listing 11.1: ScaFi-like pseudocode description (implemented in the simulation) for value-based RL algorithm applied AC. `state`, `update`, `reward` are block specific.

program structure used to combine RL with AC for improving building blocks. The branching operator (`branch`) on `learn` condition makes it possible to use the centralized training and decentralized execution (CTDE) schema since when the `learn` is `false` there is no need for a central entity (`simulation`). Here we applied a many-agent Q-Learning with hysteretic updated (from the Hysteretic Q-Learning algorithm described in Section 4.1): the Q table is gathered using `sense`, a ScaFi operator used to query sensors and collect data from them. At simulation time, Q is a shared object, but at runtime, each agent owns a local table.

Finally, the produced o_t is returned to the caller.

In this case, we encoded the learning process with AC itself. Though we could have extracted the learning process from the AC program, we took this decision because:

- it allows us to extend learning by exploiting neighbourhood Q table fields – so we can think of doing non-homogeneous learning in different areas of the system;
- the scheme for taking state and choosing actions is the same as the one we would need for learning, so the only difference is in the branch; and

- it can simply be extended to online learning.

11.4 Reinforcement learning-based gradient block

The gradient block could be generalized as follows:

```
def gradientOpt(source, metric, opt) {
  rep(infinity) { g => mux(source) { 0 } { opt(g, metric) } }
}
```

where `opt` is a function that determines how to evolve the gradient based on the field of current gradient values `g` and current `metric` (estimation of distances to neighbour). In this chapter, we examine `opt` as an *hole*, a placeholder that a reinforcement learning (RL) algorithm fills based on raw experiential interactions. The primary objective is the incremental construction of a gradient field, while also aiming to mitigate the *rising-value* issue. This issue is also known as the *count to infinity* problem in the domain of field-based coordination. It manifests as the system's inefficiency in rapidly responding to an increase in output needs (e.g., when a source node deactivates), despite its proficiency in handling situations requiring a reduction in output (e.g., when a new source is introduced).

The literature provides various heuristic solutions to overcome the limitations of traditional gradient methods [Aud+17]. One notable technique is the Constraint and Restoring Force (CRF) [Bea+08], designed to enforce a uniform rate of increase in the gradient field when nodes detect a local, slow ascent. In this context, each node is influenced by a set of constraints, specifically, nodes that possess lower gradient values. If a node identifies a sluggish increase and finds itself unconstrained, it elevates its output at a fixed rate, independent of its neighbours. Otherwise, it adheres to the traditional gradient formula. Our proposed learning algorithm aims to emulate this behaviour while eliminating the necessity for manual algorithmic design.

To articulate our learning problem, we employ the general schema depicted in Figure 11.2. The `state` function captures sufficient information for agents to dynamically accelerate local value increases. Specifically, we define the state s_t as the difference between a node's perceived output and the minimum and maximum gradient values received from its neighbours: $s_t = (|min_t - o_t|, |max_t - o_t|)$. To manage the high dimensionality of this state space, we employ discretization, thereby mitigating the risk of overfitting. The discretization is governed by two parameters: $maxBound$ and $buckets$. $maxBound$ constrains the output to a range between $-radius \times maxBound$ and $radius \times maxBound$, where $radius$ is the maximum communication range of the nodes. Values outside this range are considered

equivalent states. *buckets* defines the granularity of the discretization within this specified range.

To incorporate historical data, we stack the states from two successive time steps to form $h_t = [(s_{t-1}, s_t)]$. This composite state h_t serves as the input state for our RL algorithm, resulting in a state space cardinality of $|s_t| \times |s_t| = buckets^4$.

The action space is bifurcated into two categories: `ConsiderNeighbourhood`, which executes the traditional gradient evaluation, and `Ignore(velocity)`, which disregards neighbouring data to adjust the gradient at a designated `velocity`. The corresponding `update` function is defined accordingly:

```
def update(o_{t-1}, a_{t-1}, metric) = // o_{t-1} is the previous gradient output
  val g_classic = minHoodPlus(nbr(o_{t-1}) + metric())
  match a_{t-1} // scala-like pattern matching
    case ConsiderNeighbourhood => g_classic
    case Ignore(velocity) => o_t + velocity * deltaTime()
```

Finally, the `reward` function is described as follows:

```
def reward(o_t, simulation) {
  if(o_t - simulation.rightValueFor(mid()) ~= 0) { 0 } { -1 }
}
```

In this context, the function `mid` retrieves the corresponding field of node identifiers. The overarching goal is to compel the nodes to generate output values that closely approximate the ideal gradient values as stipulated by a trusted oracle, represented by the function `simulation.rightValuefor()`.

When the output aligns with the expected ideal value, a reward of 0 is issued. Conversely, if the output diverges from the expected value, a minor punitive measure in the form of a negative reward, -1, is administered. This is designed to expedite the nodes' convergence toward a state where the actual output closely mirrors the ideal value.

11.5 Evaluation

To evaluate our approach, we run a set of Alchemist simulated experiments and verify that an aggregate system can successfully learn an improved way to compute a gradient field (cf. the gradient block described in Part I). The source code, data, and instructions for running the experiments have been made fully available at a public repository[1], to promote the reproducibility of results.

[1] `https://github.com/cric96/experiment-2022-coordination`

Name	Values
(γ)	[0.4 – 0.7 – 0.9]
(ϵ_0, θ)	[(0.5,200) – (0.01,1000) – (0.05,400) – (0.02,500)]
(β, α)	[(0.5,0.01) – (0.5,0.1) – (0.3,0.05) – (0.2,0.03) – (0.1,0.01)]
(buckets, maxBound)	[(16,4) – (32,4) – (64,4)]

Table 11.1: Summary of the simulation variables. A simulation is identified by a quadruple (i, j, k, l) of indexes for each variable.

11.5.1 Simulation setup

The simulation comprises N devices, organized within a structured grid. The grid environments employed for this study fall into two categories, each having identical dimensions concerning width (200 m) and inter-node spacing (5 m). However, they differ in the number of rows: the first scenario features a single row (essentially aligning the nodes linearly), while the second includes five rows.

The total number of agents, denoted by N, is calculated using the formula $N = \frac{200}{5} \times \text{rows}$. Consequently, the first and second scenarios comprise 40 and 200 agents, respectively. Each node initiates its round evaluation asynchronously at a frequency of 1 Hz. The nodes positioned at the extreme left and right ends serve as source nodes. A single simulated episode has a duration of 85 s, denoted as T.

To simulate a gradually escalating issue, we deactivate the left source node at $t = 35$ s, denoted as C_s. Subsequently, the system experiences an ascent in computational field values before ultimately stabilizing.

An entire simulation spans $N_E = 1200$ episodes. For the initial $N_L = 1000$ episodes, the system employs RL to enhance a globally shared Q-table. During the subsequent $N_T = 200$ episodes, each agent utilizes the optimized Q-table, adhering to a greedy policy for action selection.

The Hysteretic Q-Learning algorithm serves as the learning mechanism (refer to Section 4.1). The behavioural policy adopts an ϵ-greedy strategy with exponential decay, optimizing the balance between exploration and exploitation. The decay rate is formulated as $\epsilon_i = \epsilon_0 \cdot e^{i/\theta}$ for each episode i.

To determine the most effective configuration, we performed a grid search across various parameters, summarized in Table 11.1. The efficacy of each configuration is ascertained by calculating the total error during the final N_T episodes, which is derived as follows:

$$error_i^t = |gradient_i^t - simulated_i^t| \tag{11.1}$$

Subsequently, the system-wide average error for each time step t is computed:

$$error_t = \frac{1}{N} \sum_{i=0}^{N} error_i^t \tag{11.2}$$

Finally, the cumulative error for each episode is given by:

$$error_{episode} = \sum_{t=0}^{T} error_t \tag{11.3}$$

We finalize the optimal configuration based on box plot analyses (Figure 11.3a), selecting the one that minimizes the average error during the last N_T episodes.

11.5.2 Results and Discussion

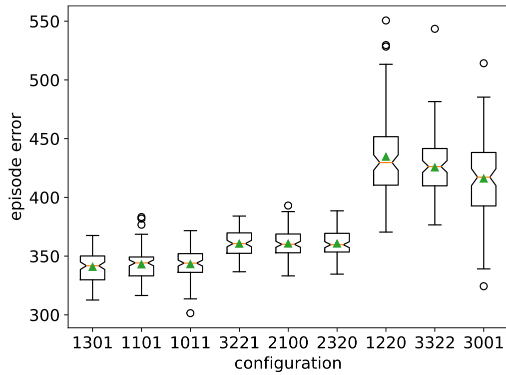
Figure 11.3 provides a comprehensive performance analysis of our reinforcement learning-based gradient algorithm. The best-performing configuration, characterized by parameters $\gamma = 0.9$, $\epsilon_0 = 0.5$, $\theta = 200$, $\alpha = 0.3$, and $\beta = 0.05$, was selected based on Figure 11.3a. The trend of the mean error across episodes is illustrated in Figure 11.3b, where the shaded region represents the standard deviation and the dashed vertical line marks the time of source alteration.

The principal objective of this research was to develop an algorithm that offers superior performance against the rising-value problem when compared to traditional gradient methods. This is corroborated by Figure 11.3b, which demonstrates that our algorithm successfully reduces the $error_{episode}$ compared to conventional approaches. Specifically, the agents adaptively learn when to disregard the neighbouring nodes and accelerate the output through the `Ignore` action.

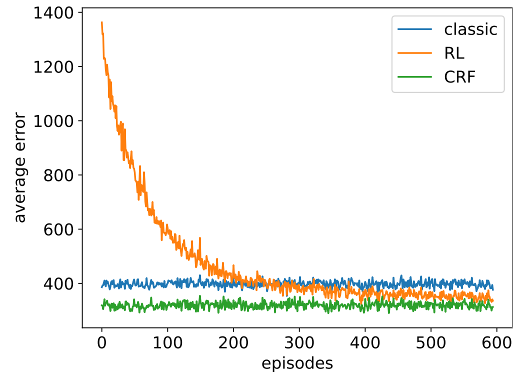
This adaptive behaviour is further substantiated by Figures 11.3c to 11.3f. Especially in Figures 11.3c and 11.3e, where the agent count is fewer, the rapid decline in error (and corresponding quick output increase) is evident when the source node is deactivated. In addition, our algorithm's performance closely parallels that of the manually crafted CRF solution for the rising-value problem. Both exhibit an initial acceleration phase followed by a transient period of overestimation, eventually leading the system to accurate gradient field values.

Another noteworthy aspect is the system-wide applicability of the learned policy. The policy is universally shared among the nodes, thereby enabling straightforward scalability across deployments with varying node counts. Moreover, the policy outperforms the baseline across different system configurations.

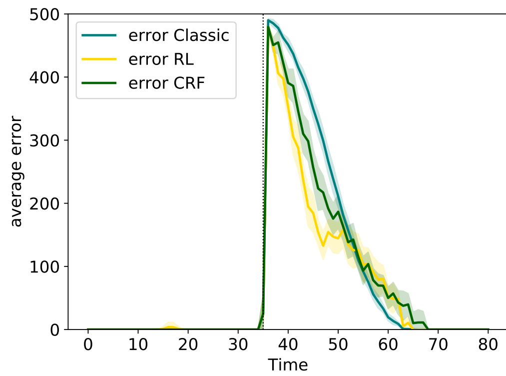
We would also like to emphasize that the asynchronous nature of node activation eliminates the need for a globally synchronized clock. This feature enhances the policy's adaptability, as it remains effective regardless of the local order in



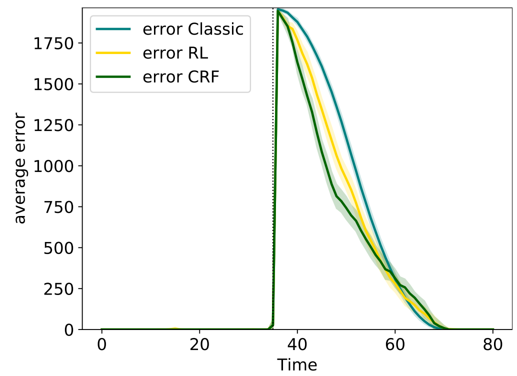
(a) Box plots of last G_s episode.



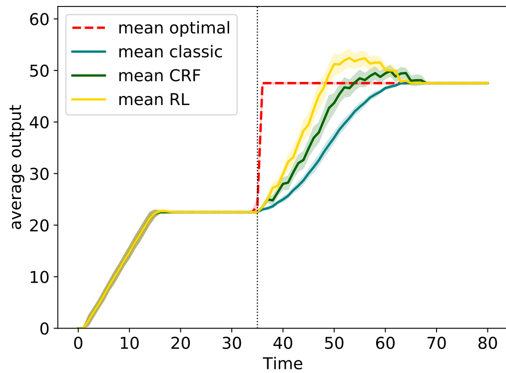
(b) Learning progress of the best result.



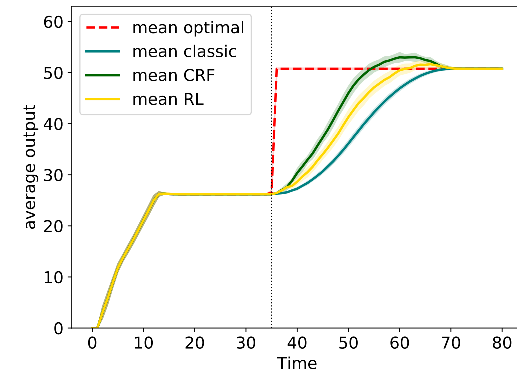
(c) Error evolution with 40 agents



(d) Error evolution with 200 agents



(e) Output evolution with 40 agents



(f) Output evolution with 200 agents

Figure 11.3: Performance of our RL-based gradient algorithm with `velocity` = 20.

which aggregate programs are evaluated. Consequently, the learned policy is ver-

satile enough to be employed in a wide range of deployment scenarios.

11.6 Final Remarks

This chapter discusses the integration of aggregate computing and reinforcement learning to foster the design of collective adaptive behaviour. In particular, we propose to use RL as a means to improve building block AC algorithms. Our approach is applied to improving the gradient algorithm, one of the key AC algorithms, where learning is performed through Hysteretic Q-Learning. We evaluate the approach through synthetic experiments comparing the reactivity of different gradient algorithms in dealing with the rising value problem. This first approach can be succinctly described as "RL for AC", where reinforcement learning is employed to enhance various facets of aggregate computing. Conversely, the next solution can be characterized as "AC for RL", in which aggregate computing techniques are utilized to inform and optimize the reinforcement learning process.

Chapter 12

Patterns: Field-informed Reinforcement Learning

How can we guide the learning of a swarm of agents to perform a collective task by leveraging a computational field?
Is it possible to use a graph neural network (GNN) to learn a policy for each agent in a swarm-like system?
– **RQ1, RQ2**

Contents

12.1 Background and Motivation 224
12.1.1 Graph Neural Networks 224
12.1.2 Problem formalization 226
12.1.3 Motivation 227
12.2 Approach Description 227
12.2.1 Architecture, fields and aggregate dynamics 227
12.2.2 Learning algorithm 228
12.3 Evaluation 229
12.3.1 Scenario 231
12.3.2 Goal 232
12.3.3 Training Phase 233
12.3.4 Test phase 234
12.3.5 Baselines 234

12.3.6 Metrics 235
12.3.7 Discussion and Results 235
12.4 Final Remarks 236

The coordination of a group of autonomous agents that can perceive and act in their environment is a fundamental problem in artificial intelligence. Such agents need to cooperate and communicate to achieve a common goal while dealing with the challenges of distributed and situated intelligence. These challenges include the limited and local nature of the information available to each agent and the emergence of global behaviour from local interactions. A key question in designing swarm-like systems is how to create distributed controllers for the agents that enable them to perform complex *collective* tasks. In this chapter, we propose a novel hybrid approach that combines the manual and automatic design of distributed controllers.

Specifically, we propose a solution called Field-Informed Reinforcement Learning (FIRL) that utilizes aggregate computing together with a GNN [Zho+20] in combination with a reinforcement learning approach, namely Deep Q-Learning (DQN). Here the GNN is trained on *field-information* from aggregate computing and provides so-called node embeddings for each agent, serving as input for the DQN. The DQN provides the appropriate actions for the agent to achieve their tasks. The main contribution of FIRL is the *definition of distributed controllers that are informed by collective knowledge that has been distilled during training, but that use only local information when deployed.* This way, FIRL can achieve a balance between manual design and automatic design, combining the benefits of both approaches while mitigating their drawbacks. Moreover, the learned policies have the potential to scale with size and adapt to different network topologies due to the inherent nature of GNNs and aggregate computations. FIRL can also be seen as a way of bridging the gap between symbolic and sub-symbolic AI methods, by integrating declarative programming with deep learning. Conceptually, FIRL leverages the ideas of *behaviour implicit communication* [CPT10] [Tum+04], whereby intelligent agents (here trained by DQN) achieve collective goals by learning how to use signs they left in the environment (here made of fields): much like ants collectively rely on pheromones they produce [Par97].

We employ this approach in a swarm-like system setting where agents are tasked to cover phenomena detected in their environment. Over time, the agents have to converge over each phenomenon to cover it appropriately as illustrated in Figure 12.1.

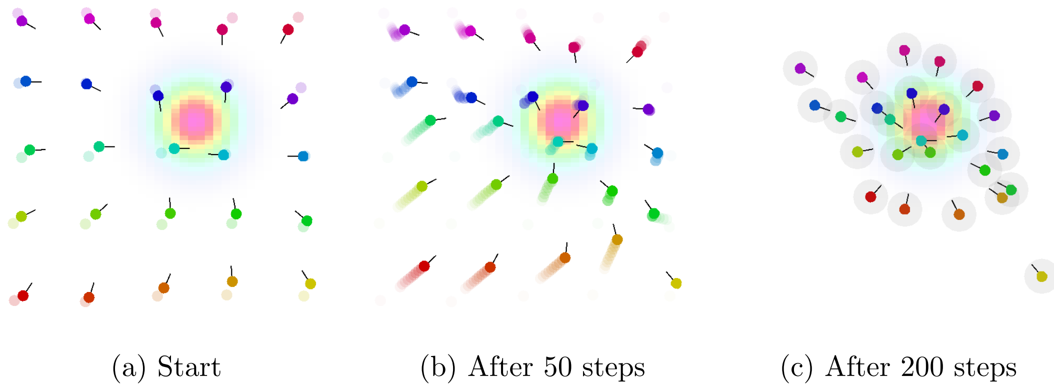


Figure 12.1: Agents (coloured dots) are deployed in an area and have to coordinate to cover the phenomenon. The phenomenon can have varying areas of importance. Over time, the phenomenon will be covered sufficiently without any central controller.

12.1 Background and Motivation

12.1.1 Graph Neural Networks

Graphs are ubiquitous structures that represent entities and their interrelationships. With the rise of deep learning, there has been a growing interest in developing neural network models that can process graph-structured data. GNN is one such model that has gained significant attention in recent years: it is a novel neural network model used to process graph-structured data with deep learning approaches. Unlike traditional neural networks that operate on fixed-size vectors, GNNs are designed to handle irregular structures, making them suitable for various applications where data is inherently graph-structured.

12.1.1.1 Graph Representation

Let $G = (V, E)$ be a graph where $E \subseteq V \times V$ defines the neighbourhood relations for each participating node, and V identifies the nodes present in the graph. Each node $v \in V$ is associated with an observation (or feature set) f_v. For the sake of simplicity, we thereafter describe G_f as a graph that contains the feature set f_v for each node $v \in V$. Note that, when we refer to G_f and G_o, we are referring to the same graph G but with different node features. Also, to access the feature set f_v of a node $v \in V$, we use the notation f_v or $G_f[v]$.

12.1.1.2 Objective of GNNs

Given f_v, the goal of a GNN is to learn the node embedding h_v for each node $v \in V$. The node embedding h_v describes the node in the network and summarizes the geometric properties of the graph in this location, allowing for the comparison of various nodes in the graph. This is analogous to how convolutional neural networks (CNNs) learn spatial hierarchies in images; GNNs learn to capture the topological and combinatorial structure of graphs.

12.1.1.3 Message Passing in GNNs

In modern GNNs, the node embedding h_v is computed by aggregating information from the node's neighbours $\mathcal{N}_G(v)$, and then combining it with the node's current embedding h_v in a process called *message passing* [Gil+17]. This iterative process allows GNNs to capture long-range dependencies in the graph.

The GNNs is partitioned into several message passing layers k, where each of them is responsible for computing the node embedding $h_v^{(k)}$ for each node $v \in V$. Formally, a GNN can be defined by three phases:

$$m_{uv}^{(k)} = \psi^{(k)} \left(h_u^{(k-1)}, h_v^{(k-1)}, e_{uv}^{(k-1)} \right) \tag{12.1}$$

$$a_u^{(k)} = \bigoplus^{(k)} \left(\left\{ m_{uv}^{(k)} : v \in \mathcal{N}_G(u) \right\} \right) \tag{12.2}$$

$$h_u^{(k)} = \phi^{(k)} \left(h_u^{(k-1)}, a_u^{(k)} \right) \tag{12.3}$$

Where h_v^k is the embedding of node v within the k-th layer, and $\mathcal{N}_G(v)$ is the set of neighbours of node v computed from E. The initial embedding h_v^0 is usually set to the node's feature vector f_v. The differential part comes into play in the ψ and ϕ functions, which are typically differentiable functions such as neural networks.

12.1.1.4 Aggregation Functions in GNNs

The ψ function, called the *message function*, computes the message $m_{uv}^{(k)}$ from node u to node v. The ϕ function, known as the *update function*, updates the node embedding $h_v^{(k)}$ of node v. The aggregation function $\bigoplus$ aggregates information from the neighbours of a node v. While simple aggregation functions like sum, max, or sum of products are common, more complex aggregation functions have been proposed to capture intricate relationships in the graph [Pel+20].

12.1.1.5 GNN Application

Applying a GNN to a graph G_f can be expressed as:

$$GNN(G_f) = \{h_v^{(k)} : v \in V, k \in \mathbb{N}\} \tag{12.4}$$

This formulation allows GNNs to effectively process and extract features from graph-structured data by iteratively aggregating and transforming information from the node's neighbours.

GNNs have found applications in diverse areas. They are used in social network analysis to understand user behaviours and community structures. In chemistry, they help in predicting molecular properties and drug discovery. In physics, they assist in understanding complex systems. In this chapter, we delve into the application of GNNs in multi-agent systems, focusing on learning local behaviours for each agent (more details in Section 12.2).

12.1.2 Problem formalization

Given the homogeneity, large system scale, and the *locality* (i,e., each agent can only observe its neighbours), the problem can be modelled through the SwarMDP model [Sos+17]—an extension of the decentralized partially observable Markov decision processes (DecPOMDP) [BZI00] model for swarm-like systems (see the Part I section for more details). However, In swarMDP, the neighbourhood is not directly defined, but it is implicitly defined by the observation model ξ. In our specific case, the agents can only interact with 1-hop neighbours and are not directly influenced by other agent observations. We can therefore restrict the observation model as follows:

$$\xi(v) : \{s_j, j \in \mathcal{N}^v\} \to O \qquad \xi = \{\xi(v), v \in \mathcal{P}\}$$

where $\mathcal{N}^v$ is the set of neighbours of v. This model can be used to express the evolution of the system in time. Specifically, starting from a global state $\mathcal{S}_t^P$, the next state $\mathcal{S}_{t+1}^P$ is defined as:

$$\mathcal{A}_t^P = \pi(\xi(S_t^P)) \qquad \mathcal{S}_{t+1}^P = \mathcal{T}(\mathcal{S}_t^P, \mathcal{A}_t^P)$$

Given a time t, the system can be also represented as a graph $G^t = V^t, E^t$, where E^t is built from $\mathcal{N}$. Each node is then decorated with the local observation perceived at the time t: $o_t^v \in O$. This graph can be used both to compute computational fields and, as done in previous work [Tol+19; Tol+20; Gos+22], can be the input for a GNN.

12.1.3 Motivation

Our work uniquely intersects the realms of automatic and manual methodologies, with a focus on field-based coordination and ManyRL empowered by GNNs. Building upon foundational research in co-fields [MZL04], we advance a novel framework where agents utilize a “digital sign” or computational field [CPT10]. This enables them to access and reason over global system data, thereby enhancing their decision-making capabilities.

In our innovative model, we integrate ManyRL and GNNs to foster agent intelligence. This architecture allows agents to learn localized, yet comprehensive, representations of their surrounding environment.

Our approach diverges significantly from existing studies in which GNNs function as part of a distributed controller [Gos+22; Tol+19]. Although these works have demonstrated GNNs' utility in decentralized systems, they often relegate the communication aspect solely to neural networks. This has been known to complicate and potentially destabilize the learning process.

In contrast, our implementation ensures that the GNNs are guided by computational fields, narrowing the learning focus to specialized tasks as outlined by a collective reward function. This design not only expedites learning but also enhances its stability.

12.2 Approach Description

In this section, we discuss the components involved in our proposed solution—namely the *architecture*—and how these components interact with each other to bring the system to perform the collective behaviour—namely the *dynamics*. Finally, we will detail the learning algorithm designed and used to synthesize the policy.

12.2.1 Architecture, fields and aggregate dynamics

The proposed solution, summarized in Figure 12.2, consists mainly of two parts:

- the aggregate program used to create part of the observation;
- the policy π_{gnn} learned through GNN-based approach.

Let Γ be the aggregate program that takes a graph G^t decorated by o_t^v, representing the participating agents and their neighbourhood relations at time t, as input. The evaluation of Γ produces a field value θ_t^v for each node v in G^t. From this field, we construct the feature vector f_v for each node v in G^t as follows: $f_v = (\theta_t^v, o_t^v)$. The policy π_{gnn} is then evaluated for each agent using f_v as input, producing an

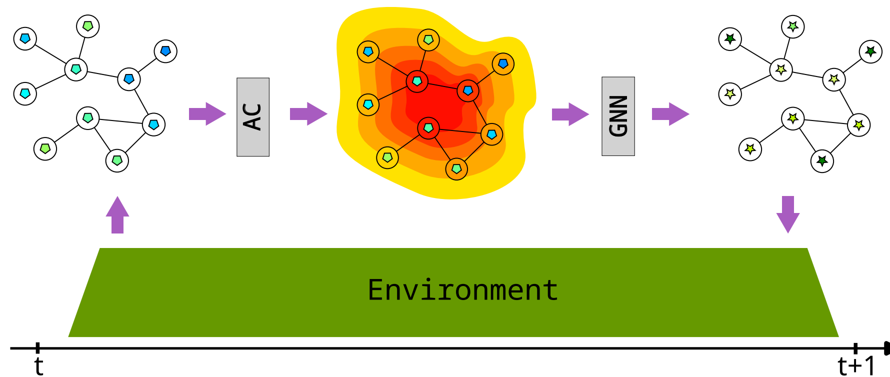


Figure 12.2: High-level description of FIRL approach. For each time step t, a graph is constructed from the environment, and it will associate each node with a local feature o_t^v (hexagons in the picture). Using this feature, an aggregate program computes spatio-temporal information that enhances the feature set of each agent producing f_v, depicted as colours in the middle graph. Finally, utilizing the GNN, actions are computed for each agent in the system to be performed against the environment, enabling advancement in the simulations according to swarMDP rules.

action a_t^v that will then modify the global state of the system. While the graph, containing the aggregate information, might appear as global knowledge, this is not the case as the information is *never* aggregated globally. The individual agents only combine information from their local neighbourhood. In fact, the program Γ is proactively executed at every agent, and the GNN can be locally evaluated using only neighbourhood information. We want to emphasize that, in this case, the GNNs must be 1-hop; otherwise, they could not have a local interpretation for each agent, according to our system model. This intuition of basing a policy based only on neighbourhood information came from mean-field RL, introduced in Chapter 4

12.2.2 Learning algorithm

Here we discuss a variant of many agent Deep-Q learning: we use a *value-based* approach combined with a GNN as a function approximator. Specifically, we leveraged the property of GNNs to have a *dual* interpretation, i.e., to function globally over the entire graph and locally only over the neighbourhood. Importantly, each agent only has local information from itself and its neighbourhood to utilize in the GNN. The major modifications to many-agent DQL are (see Section 4.1):

1. experience replay stores experiences in the form of *graphs* decorated with features (e.g., observations, actions, rewards, etc.),

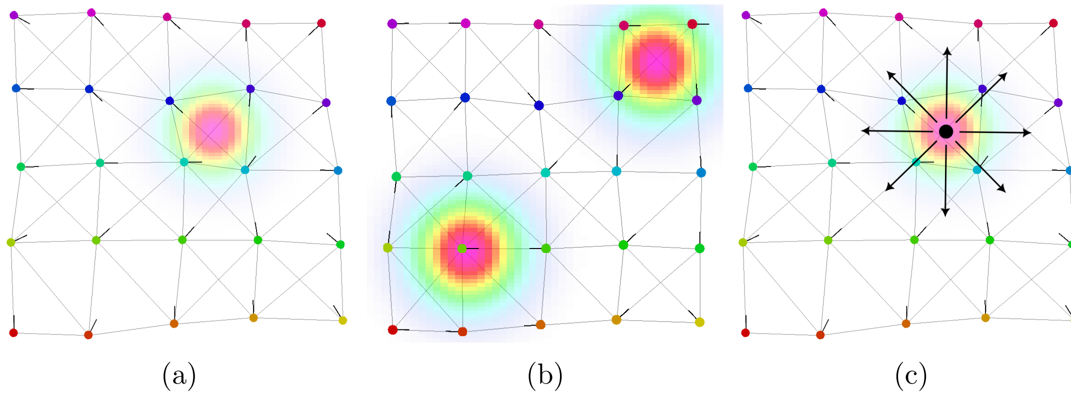


Figure 12.3: Simulations of the case study scenario in Alchemist. The dots represent drones, the circle area represents the phenomenon to be monitored. Figure 12.3a represents the scenario used during training as well as during test. The others instead are only used in the test phase to evaluate the policy found.

2. the neural network used to compute the Q function is based on a GNN with an multi-layer perceptron (MLP) downstream.

The first point is a natural extension because we work on graphs rather than simple values. This also influences how we create a batch of experiences to train the network. In fact, we sample a batch of graphs from the replay buffer, and then we merge them into a single graph, which is then used to train the network. This process is called *graph mini-batching* [FL19; Wan+19] and its main purpose is to pass an entire batch of graphs to the same GNN for improved performance. For the second point, the use of GNNs allows us to define policies on a variable neighbourhood, which is essential in such systems as this can change due to the applied neighbourhood policy. It is known that GNNs have a certain ability to generalize to new structures and scale with different agents [Zho+20; KTA19]. Additionally, using the overall graph compared to local experiences makes learning more stable as it reduces the non-stationarity of the environment perceived by each node. This is because, even though the actions are produced using only local and neighbourhood information, during the learning phase, we have access to the internal graph, which will influence the policy through non-local information during the backpropagation.

12.3 Evaluation

To test the effectiveness of the proposed approach, we experiment with a case study related to swarm robotics leveraging Alchemist, specifically, tracking and

Algorithm 3: Deep Q-Network (DQN) with GNN and Graph Replay Buffer executed by each agent

Input: Environment $\mathbb{E}$, graph replay buffer $\mathcal{D}$, target network θ^-, current network θ, exploration strategy ϵ
Output: Trained DQN model θ
Initialize $\mathcal{D}$ with random initial transitions;
Initialize θ with random weights;
Set $\theta^- \leftarrow \theta$;
while *not done* **do**
  Observe current graph observations G_o;
  if *random* $< \epsilon$ **then**
    select a random action a;
  else
    $G_q = Q(G_o, \theta)$; $a = \{v \in G_q | a_v \in argmax_{a_v} G_q[v](a_v)\}$;
  end
  Execute the collective action G_a in the environment $\mathbb{E}$ and observe a graph-level reward G_r and the next observation G'_o;
  Store transition (G_o, G_a, G_r, G'_o) in $\mathcal{D}$;
  Sample a batch of graph transitions $(G^i_o, G^i_a, G^i_r, G'^i_o)$ from $\mathcal{D}$ and merge them in $(G^b_o, G^b_a, G^b_r, G'^b_o)$;
  Compute the target Q-value for each node v in the graph G^b:
  $y_v = G^b_r[v] + \gamma * \max a' Q(G'^b_o[v], G^b_a[a']; \theta^-)$;
  Compute the current expected value for each node v in the graph G^b:
  $y^*_v = Q(G^b_o[v], G^b_a[v]; \theta)$;
  Update the current network weights using gradient descent:
  $\theta \leftarrow \theta - \alpha \nabla \theta \frac{1}{|G^b|}(y - y^*)^2$;
  Every C steps, update the target network weights: $\theta^- \leftarrow \theta$;
end

coverage of a spatio-temporal phenomenon—cf. tracking a wildfire or monitoring the water levels in a canal with multiple autonomous drones embodied in embedded devices (e.g., drones or IoT devices)—similar to the ones discussed in Part I. The individual drones do not have any knowledge of the initial phenomenon itself (i.e., shape, size, location, velocity, and so on). Initially, we perform the training phase using a stationary phenomenon before expanding towards a moving phenomenon in the test phase. Finally, the phenomenon may have varying areas of interest, defined by an underlying distribution function. This underlying distribution is utilized in the feature set of each drone's observation and guides the drones to rally over the phenomenon. While we only use Gaussian distributions, we can use any other distribution and shape. For the GNN, we use the implementation provided by PyTorch Geometric [FL19], which is a library for deep learning on graphs built on top of PyTorch [Pas+19]. Finally, we use ScaFi [Cas+22b] as the aggregate programming language to support our field-informed approach. The evaluation is performed in two stages, first, the neural networks are trained in an explicit training phase before being extensively evaluated in the testing stage. [1]

12.3.1 Scenario

Figure 12.3 presents the three different types of scenarios utilized within the evaluation. The first type of experiment considers a single phenomenon at a static location (i.e., *Zone Fixed*), the second type of experiment considers two phenomena in two independent but static locations (i.e., *Two Zones*), and the third type of experiment considers a moving phenomenon (i.e., *Moving*). All phenomena are modelled as a Gaussian distribution. Importantly, only the left scenario illustrated in Figure 12.3a was used for training the neural networks. Furthermore, all three types of scenarios contain a set of $\mathcal{P} = 25$ drones placed in a 2D grid large 1000x1000 meters. Each drone can perceive the presence of the phenomenon of interest through an installed sensor ζ_v with $v \in \mathbb{V}$. (e.g., camera, temperature sensor, etc.) if it is within range. Additionally, each drone has a coverage range ω (fixed to 75 meters) that describes the area it can monitor. Each drone can only communicate directly with its neighbourhood $\mathcal{N}$, which in this case depends on an $\mathbb{O}$ range fixed to 300 meters. Through this communication channel, drones can exchange information. Each drone moves following a certain action composed of two components (r, i) which respectively describe the angle and intensity of the movement (i.e., the velocity vector). Since we used a value-based approach, the action space $\mathcal{A}$ is discrete and composed of 18 possible angles and 3 possible intensities. In particular, the angles are quantized to 20 degrees, and for the velocities,

[1] The simulations are publicly available at `https://github.com/AggregateComputing/experiment-2023-acsos-field-informed-rl`.

we have selected [0, 5, 10] m/s.

For the aggregated information, each drone will produce a computation field with which they will try to approximate the direction of the phenomenon of interest. The program Γ in question is a simple application of block `G`, where the source is the maximum value of the neighbourhood. This can be expressed in ScaFi as follows:

```
val source = maxHood(nbr(sense(ζ))) == sense(ζ)
G(source, Point3D.Zero, _ + nbrVector())
```

where `maxHood` is a function that returns the maximum value of the neighbourhood, `nbr` is a function that a neighbourhood field of values (in this case, the sensor value ζ), `sense` is a function that returns the value of the sensor, and `nbrRange` is a function that returns an approximate direction for each drone in the neighbourhood. This value will then be fed into the π_{GNN} to compute the action to be performed.

12.3.2 Goal

The objective of this scenario is threefold:

1. maximize the number of drones within the phenomena;
2. minimize the number of drones without neighbours;
3. maximize the coverage of the system.

As we are modelling a reinforcement learning system, these three components must be encoded in a reward function that provides an estimate of the current action taken by a given drone. Formally, we define the reward of a drone being within the phenomenon as:

$$R_v^a = 1 \text{ if } \zeta_v > 0 \text{ else } 0 \tag{12.5}$$

Namely, a drone is considered within the phenomena as soon as the drone can sense the phenomenon. This will lead the system to prefer a configuration in which every drone is present within the phenomenon. The second element in the objective function ensures cohesion among the drones. This is important because if the system breaks into many scattered drones, the observability of the phenomenon is reduced, limiting the ability of the drones to move appropriately in the environment. In this case, the reward is defined as:

$$R_v^{\mathcal{N}} = 1 \text{ if } |\mathcal{N}| > 0 \text{ else } 0$$

Finally, to maximize the coverage, we define a reward function that favours the maximum distance between the drones equal to the coverage range ω. This will

minimize multiple drones covering a common area. This means that the average distance will tend towards the one expressed by the viewing range of each drone:

$$R_v^C = 1 - \frac{d_{min}}{\omega}$$

Where d_{min} describes and minimum distance of the drone to its neighbourhood. The final reward function R_v for an agent v is defined as:

$$R_v = (\frac{R_v^a + R_v^{\mathcal{N}} + R_v^C}{3}) - 1$$

Specifically, we decided to express the signal as a *regret* as it is a more general measure of the quality of the action taken by the drone.

12.3.3 Training Phase

Before we can evaluate our approach, the underlying neural networks have to be fine-tuned in a dedicated training phase. The training process for each neural network was divided into 100 episodes, each consisting of 200 steps, resulting in a total of 20,000 experiences. For each episode, the 25 drones are semi-randomly positioned on a grid (i.e., in a lattice layout with a random variation in their position) without knowing the correct position of the phenomenon, but that is fixed in the top right corner. The position of the phenomenon with an example of positioned drones can be seen in Figure 12.3a.

The feature set used by the GNN created for each drone consists of the vector computed by the aggregated program and the value of the local sensor ζ. In this case, we chose to use an exponential epsilon decay, defined as:

$$\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \cdot e^{-\lambda \cdot e}$$

Where e is the current episode number. This leads to a high number of random actions at the beginning and gradually shifts towards exploitation in the later episodes. In our training process, we set $\epsilon_{min} = 0.02$, $\epsilon_{max} = 0.99$, and $\lambda = 0.1$. γ was set to 0.99, as we want to give more value to future returns, aiming to achieve good coverage by continuously tracking the phenomenon. The neural network structure used consists of a layer of SuperGAT [KO22] —a GNN based on attention mechanisms— and a layer of MLP. The hidden size was set to 256. As the reward function is defined as a regret, we decided to use the Huber loss function with $\delta = 1$. which is a combination of the L_1 and L_2 loss functions:

$$L_\delta = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| < \delta \\ \delta \cdot (|y - \hat{y}| - \frac{1}{2}\delta) & \text{otherwise,} \end{cases}$$

where δ is a hyperparameter that determines the threshold between the two loss functions (fixed to 1 in our experiments) and y and $\hat{y}$ are the target and predicted values, respectively. This function is used to penalize the drone if the action taken is too far from the optimal action. We use the RMSprop optimiser with a learning rate of 0.0001. Finally, we use a replay buffer of size 1000 to store the graph experiences and a batch size of 32.

12.3.4 Test phase

For the evaluation, we explore the previously discussed three different types of experiments. We generated 64 random scenarios for each type of experiment. Additionally, the placement of the drones was randomized as it has been done during training. For the first type, consider a single static phenomenon randomly placed in the environment. This is in contrast to the training where the phenomena were always placed in the same location. For the second type, we placed two distinct phenomena within the area. Their location is kept constant in all 64 experiments. As the training only contained a single phenomenon, this setup represents a challenge for the drones. Finally, the third type contained moving phenomena. In each scenario, the starting position as well as the direction of movement is randomly sampled from a uniform distribution. The movement is in a straight line with a constant speed of 5m/s within an unbounded environment. Examples of all three types of experiments are shown in Figure 12.3.

12.3.5 Baselines

We compare our FIRL approach against baseline approaches where the DQN utilizes a MLP as well as an approach only relying on GNNs, without additional field information. In all approaches, the underlying neural network (i.e., the MLP and the GNN) are trained with a single, stationary phenomenon.

The MLP uses the same feature set as the GNN but applies it in the DQN but without leveraging the graph structure. Moreover, we increased the batch size to 512 and the replay buffer to 10000 since we recorded 25 drones' experiences for each step instead of one graph experience. The GNN alone, without using the field information, apply the position of drones and the local sensor value directly as input features within the DQN. These baselines are used to verify the effectiveness of the components used in the FIRL. Indeed, the MLP baseline is used to verify the effectiveness of the GNN in the proposed approach, while the GNN baseline is used to verify the effectiveness of the field information in the proposed approach.

12.3.6 Metrics

We evaluate the performance of the different approaches by measuring the coverage of the phenomenon over time. The coverage is defined as the percentage of the phenomenon covered by drones. Specifically, we can measure the coverage as the intersection over the union of the phenomenon and the drones' view range. Formally, we define the overall coverage for a certain time step as:

$$\Omega = \bigcup_{v \in V} \omega_v \quad C = \frac{|\Omega \cap \mathcal{P}|}{|\mathcal{P}|}$$

where $\mathcal{P}$ is the area of the phenomenon and Ω is the area covered by the drones. In the training phases, we measure the average coverage in each episode, and the total reward obtained by the drones at each episode of the simulation. We also measure the number of drones that are within the phenomenon at each step of the simulation. This will be a measure of how well the drones are tracking the phenomenon.

12.3.7 Discussion and Results

The results of the training process are summarized in Figure 12.4. In the charts, the line represents the average value of a metric of interest, while the shaded area represents its standard deviation. Specifically, we observe that the proposed version achieves higher coverage and total reward compared to other approaches. Interestingly, despite the global information available in GNNs without fields, they fail to converge to a good result like the one obtained with the field. This outcome was expected, as the computed field helps drones encode the necessary information to navigate towards the phenomenon. Furthermore, we note that GNN combined with DQN and graph replay buffer outperforms the simple MLP informed field computation. This is because relying solely on MLP and basic deep learning leads to non-stationary and unstable learning, as evident from the wider confidence interval of the reward over training time.

Focusing now on the results of the test phase, highlighted in Figure 12.5, we observe that the field-informed version achieves higher coverage than the other approaches in all scenarios since it shows the ability of our solution to generalize to situations. We observe that the field-informed version successfully moves the drones closer to the target phenomenon, distributing them evenly without collapsing into a single central point. Figure 12.5 quantitatively presents the results across various previously described scenarios.

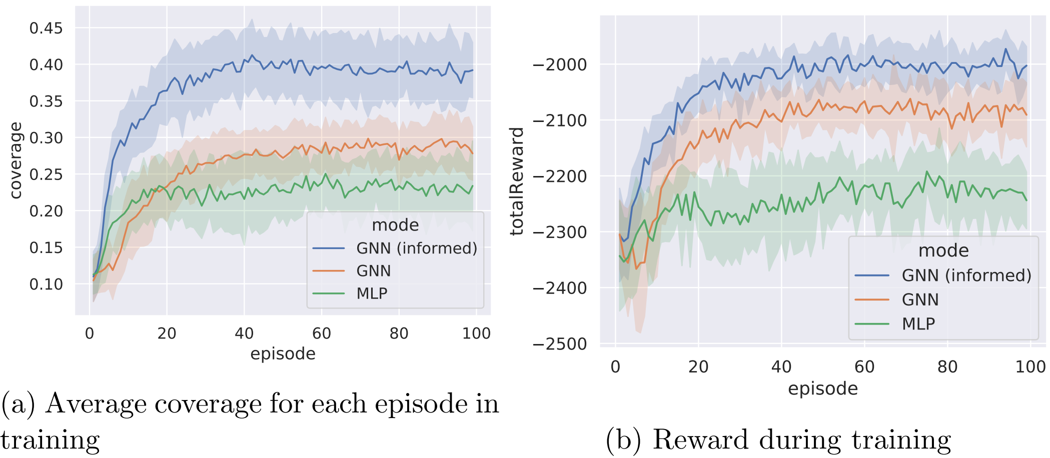
For all experiments, both GNN versions demonstrate the capability to transfer the learned experience to the test phase whereas the MLP version fails to generalize. It is worth noting that, in the *Zone Fixed* experiment, once the desired

configurati45on is achieved in the static case, the drones cease to move, maintaining the found configuration. Interesting observations arise when we use scenarios different from the training phase. In the *Two Zones* experiment, we notice that our approach using field information finds a better configuration than the simple GNN counterpart. It exhibits both higher overall coverage and manages to divide the system into two equally covered parts. Indeed, observing the Figure 12.6, we notice that the informed version maintains a balanced coverage between the two zones, with a difference of less than 5% between the two parts, maintaining a fair division of the phenomena. In contrast, the uninformed version also maintains a fair division but with significantly different coverage between the two parts, indicating a wrong placement among the drones in one of the zones. This is a consequence of the uninformed version's inability to encode the necessary information to divide the drones into two zones, therefore it is not able to generalize. Finally, the *Moving* experiment emphasizes how the informed version generates a more robust policy for new scenarios. Indeed, we observe that our approach using FIRL maintains higher coverage and a greater number of drones on the target phenomenon compared to the other two solutions. The uninformed GNN version, however, fails again to generalize its movement behaviour, as evidenced by the simulations where the drones, once reaching the target zone, stop moving due to tracking issues.

In conclusion, the results demonstrate how the proposed idea can generate more robust controllers. By guiding information flow in GNNs, we improve learning efficiency and alleviate the challenge of encoding relevant information. Nevertheless, we acknowledge the crucial role of GNNs. Our modified version of DQN, combined with GNNs, enables the discovery of robust behaviours in a few episodes, which is challenging to capture with MLPs combined with DQN, even if we use field information.

12.4 Final Remarks

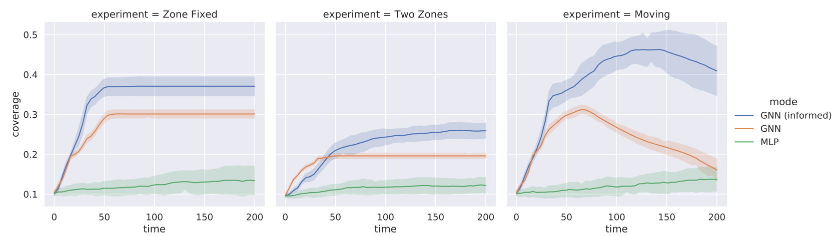
In this chapter, we have introduced a novel approach for constructing distributed controllers by leveraging aggregate computing to encode agent interactions, along with the combination of DQN and GNN for synthesizing distributed intelligence. The proposed *Field-Informed reinforcement learning* (FIRL) approach offers a promising solution to the challenges faced in coordinating multi-agent systems. By combining manual design and machine learning techniques, the approach enables agents to autonomously learn and adapt their behaviour while leveraging locally available information. The demonstrated success in the proposed case study in solving collective tasks underscores the potential impact of this approach in advancing the field of multi-agent systems and swarm robotics. In the next chapter,



(a) Average coverage for each episode in training

(b) Reward during training

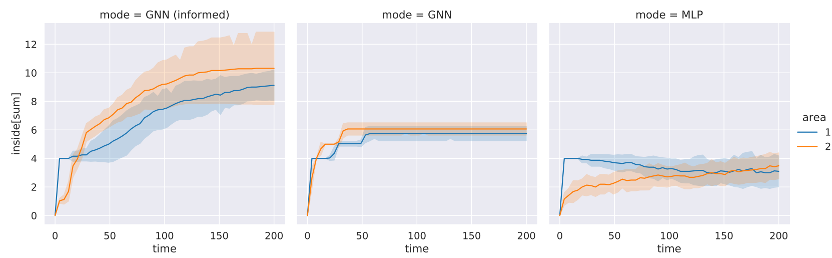
Figure 12.4: Training results of FIRL. It can cover the phenomenon better than the baselines, and it reaches a higher reward.



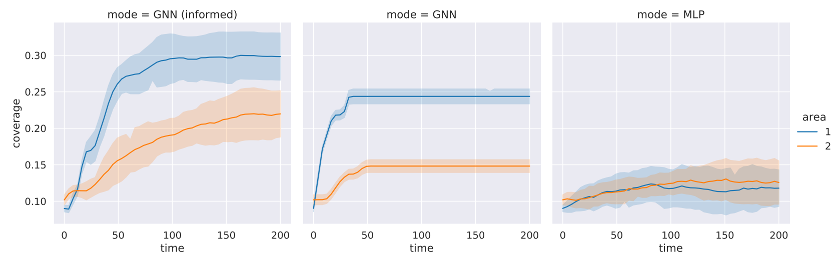
(a) Ratio of coverage of the phenomena. Our FIRL approach can outperform other approaches lacking field information.

(b) Number of drones inside the phenomenon in the three types of experiments

Figure 12.5: Quantitative test results. The proposed approach can cover and track the phenomenon better than the baselines.



(a) *Two Zones* experiment: aggregated number of drones inside each phenomenon



(b) *Two Zones* experiment: ratio of covered to the uncovered zones both phenomena

Figure 12.6: Coverage of two zones using the different modes of the controller.

we will discuss how reinforcement learning can be used to platform aspects of the aggregate computing model.

Chapter 13

Platform: Distributed Schedulers for Collective Computations

Is it possible to improve the efficiency of aggregate computations without modifying the application logic?
– **RQ2, RQ4**

Contents

13.1 Background and Related Work 240

13.2 Aggregate Platform Improvement Through Reinforcement learning 241

13.2.1 Learning Setting 242

13.2.2 Reinforcement learning to Reduce Energy Consumption 243

13.3 Evaluation 244

13.3.1 Simulation Setup 245

13.3.2 Discussion and Results 247

13.3.3 On practical applicability 251

13.4 Final Remarks 252

An aggregate program is usually deployed across a network of devices that interact with their neighbours. These devices execute the program through asynchronous sense-compute-act rounds [Cas+20b]. The goal is to facilitate the emergence of collective behaviour through repeated evaluations of the program, thereby directing the system towards globally coherent objectives. The program encompasses both *functional* and *non-functional aspects*, such as resilience and resource

complexity in terms of time, memory, and message overhead. Some of these non-functional aspects can be managed at the middleware level. The aggregate computing (AC) execution platform serves as the logical or software middleware that coordinates networked devices, enabling them to operate as an aggregate system and supporting the execution of aggregate applications through properly scheduled computation rounds [Pia+21]. Typically, the aggregate platform is manually configured to initiate rounds based on a specific frequency or triggers. However, this chapter investigates the application of RL at the platform level to discover effective scheduling policies. The aim is to enhance system efficiency while preserving its functionality, as characterized by the eventual collective outcome.

We substantiate our thesis through a case study, where a variant of many-agent Q-Learning is successfully applied to minimize system-wide power consumption without requiring modifications to the aggregate program itself [LR00].

13.1 Background and Related Work

The problem we deal with in this work may fall under the topic of Multi-Objective Sequential Decision-Making [Roi+13]. In fact, our goal is to optimize a functional goal (e.g. crowd steering) and one or more non-functional goals (e.g. reducing energy consumption, increasing the speed of calculation, and others).

In particular, these goals may also conflict, making it difficult to find the optimal policy for a given problem. Particularly, we focused on applying RL in the case of the trade-off of functional and non-functional concerns, since it has been already exploited for managing non-functional aspects in several applications - e.g., reducing the energy in a smart building [Yu+21], improving the efficiency of routing protocols [HCL18] and improving cooling in server farms [Le+19]. Moreover, RL is practically the first choice as it allows learning directly from raw experience without the need to build an explicit model with minimal overhead at inference time (both in the case of deep neural networks and standard tabular methods).

The focus of our solution is mainly on energy efficiency related to self-organising computations—since it is an important topic nowadays related to green computing concerns. One way to improve efficiency would be to act at the level of scheduling as already explored in related work about wireless sensor network scheduling. In [Iwa+21; Mih+12] the system learns when a node should be awakened to reduce conflicting messages (and therefore to reduce the power consumption). However, our work is quite different from the previous. We aim to leverage learning to improve a *general* collective computation expressed in AC-like execution model (i.e, asynchronous and iterative evaluation of rounds eventually lead to collective behaviours–see Part I) and not a specific application as is often the case (e.g. distributed sensing in WSN). For example, consider a typical AC application,

namely the management of crowds of people equipped with smart devices. The self-organising execution model involves fixed and periodic evaluation of rounds. But this means that at unhurried times with few emergencies, the system continues to compute the same value without adding any collective information. Therefore, with RL we want the system to learn to identify these "peaceful" conditions and then reduce the number of collective rounds.

Another essential non-functional aspect that can be addressed using RL is the management of message bandwidth. Numerous studies have explored efficient communication protocols [ZZL19; Su+19]. For computations resembling the AC model, nodes are required to communicate with their neighbours using broadcasts to eventually achieve a collective data structure. However, in many scenarios, this approach can result in unnecessary message transmissions. For instance, in highly dense networks, transmitting local information to only a select group of neighbours might be sufficient to establish the same global structure. Consider the scenario of crowds as an illustrative example. In an extremely dense setting, if a node intends to disseminate an alarm throughout the system, it might be more efficient to relay the message to just a subset of its neighbours, adhering to a gossip-like protocol. In such contexts, RL can be instrumental in determining when nodes should modify their neighbourhood model, thereby reducing the number of messages transmitted while still achieving the intended application objective.

13.2 Aggregate Platform Improvement Through Reinforcement learning

In AC, collective behaviour is the result of both an *aggregate* program and an *execution* protocol whose details may vary within certain limits without affecting the desired functionality. The extent of effects induced by the execution protocol is usually assessed by *simulation testing*, possibly accompanied by formal results applicable to the aggregate program at hand—e.g., self-stabilisation [Vir+18] or known properties (cf. optimality results) of used algorithms [Aud+17; Aud+21].

Programming in AC focusses primarily on *functional* aspects of an application, but may also pay special attention to non-functional aspects like, e.g., *resilience* (supported by self-organization algorithms). Other non-functional goals such as computational efficiency and bandwidth consumption may also be important (cf. energy-constrained devices), and can be addressed through (i) *efficient algorithms*, in terms of computation, memory, or message complexity, as well as (ii) fine-tuned configuration of the *execution platform*, namely of the software middleware, daemon, or simulator responsible for supporting the execution of aggregate applications. For instance, the designer could configure the platform to schedule

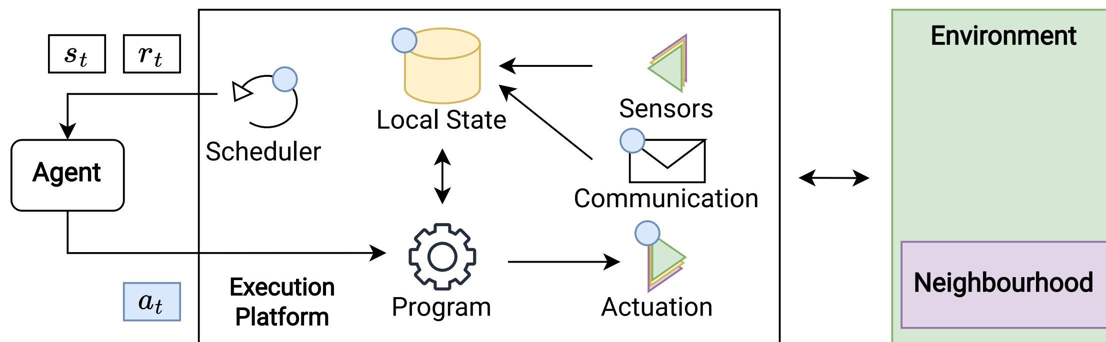


Figure 13.1: Description of the general scheme of RL applied to the execution platform. The agent receives the state and reward from the platform and produces an action that can affect one of the platform aspects (blue circles).

computation rounds at a given frequency that is somewhat related to the desired reactivity and expected environmental dynamics.

Recent work [Pia+21] has also proposed the use of rule-based and reactive policies to let a system *adapt* its execution based on changes in inputs or the environment. However, such rules are still hand-crafted: they may be highly suboptimal or require to be adjusted when porting the application to different environments.

In this chapter, we explore the possibility of *learning* how to improve aggregate applications from a non-functional point of view, leveraging the RL framework. Particularly, in our idea RL enhances general aggregate programs, making it possible to reuse the same application logic in different contexts. This choice has several benefits. Firstly, learning will be used as a mechanism to improve *adaptability*. Indeed, using hand-craft approaches to handle non-functional aspects could lead to ad-hoc solutions that might fail against environmental changes not considered by the designer. Using learning instead, the agents self-adapt as a consequence to maximize the reward signal that guides them towards a collective goal. Also, in the case of *online* and *continuous* learning, it is possible to learn a good policy *by doing*, even if the nodes are in an unknown environment without any prior knowledge of the application domain. This situation is difficult to handle with handcraft algorithms, that require a deep domain knowledge.

13.2.1 Learning Setting

The learning settings follow the one discussed in Section 4.3.2, therefore a many-agent Q-learning configuration with *parameter* and *experience* sharing. The actions chosen by the agents only indirectly influence the local program output: this is done by producing side-effects at the infrastructural level—e.g., dropping ob-

solete messages from neighbours, or waking up more frequently to more quickly react to locally perceived changes. The actions can also influence the neighbourhood state, e.g., by sending a special message to force neighbours to wake up. This work considers a *local* reward function, that should be crafted in a way that brings about the collective goal through emergence. However, the reward definition is specific to the *non-functional requirement* that is being considered; in other cases, there might be the need to design more complex reward functions, such as global functions (i.e., the reward signal is received after collective actions) or neighbourhood-level functions (i.e., the signal is received after a neighbourhood-level action). Nevertheless, for the requirement taken into account (e.g., energy efficiency—see next subsection), the reward function can be reused across several collective program specifications, since it does not depend on the application logic.

13.2.2 Reinforcement learning to Reduce Energy Consumption

Among the non-functional goals that RL could face, this study focusses on reducing the *energy consumption* of aggregate computations, by altering the local agent *scheduling policy* (As pointed out in Section 13.3.2). Our goal is to reduce the number of computation rounds and hence the energy needed to achieve certain results in a certain time amount or, dually, to reduce the amount of time to achieve certain results for a given amount of energy. Similar ideas have already been considered in related works about wireless sensor network scheduling. In [Iwa+21; Mih+12] the system learns when a node should wake up in order to reduce conflicting messages (and therefore to reduce the power consumption). However, our work is quite different from the previous. We aim to leverage learning to improve a *general* collective computation expressed in AC-like execution model (i.e., asynchronous and continuous evaluation of rounds brings to collective specifications).

In particular, the algorithm should learn how to reduce the round frequency in stable conditions. To this aim, the program should be self-stabilising [Vir+18], i.e., it should reach a well-defined eventual fix-point field result, once input fields cease to change. Note that, by reducing the round frequency, we reduce both the total amount of program evaluation *and* the message exchange between neighbours (which typically involves non-negligible power consumption).

Even if we consider the whole aggregate computing context as a state, in this work the agent observation space is based on the local output produced by an aggregate program evaluation. Following the self-stabilisation assumptions, an agent state encodes the variation of the output history, thus we constrain the output to be a numeric value. At each time step t, it is computed the local output o_t and δ_t, which consists of three possible values, `Stable` ($o_t = o_{t-1}$), `Rising`

$(o_t > o_{t-1})$, and `Decreasing` $(o_t < o_{t-1})$. Also, in order to consider the evolution of δ, each agent stacks the last w values of δ in its state: $s_t = (\delta_t, \delta_{t-1}, \dots, \delta_{t-w})$.

The actions point out when the agent should fire the next aggregate program evaluation, following a typical wake-up scheduling. We consider a discrete action set, e.g., based on possible energy consumption profiles. Each action contains the delta time at which the next round should be triggered.

Finally, the reward function is devised by only observing the local state of the AC execution platform, aiming to reduce the overall consumption by emergence. The reward signal should consider two aspects:

- the overall low-power consumption;
- the time needed to reach a stable condition.

In doing this, the signal weighs these two contributions using an additional parameter θ. When the output history contains a $\delta \neq$ `Stable`, the reward function is evaluated as:

$$r_t = -\theta * \Delta/T \tag{13.1}$$

T is defined by the action with the highest next wake-up value. This gives a negative reward if the node stays in a non-stable condition for a long time (i.e., when $\Delta = T$). Otherwise, the reward function is evaluated as:

$$r_t = (1 - \theta) * (1 - \Delta/T) \tag{13.2}$$

That is the inverse of the case of non-stable conditions. Thus, this reduces the consumption as much as possible, so the reward is maximized when $\Delta = T$.

Notice that these settings do not depend on a particular aggregate program, but they can be used in any program with continuous output and an eventually stable field. Besides, thanks to the local reward function, this learning setting could be also employed for online learning. However, we would highlight that in this case, we exploited offline learning. In this way, we could consider the cost of RL at runtime negligible concerning power consumption.

13.3 Evaluation

Our RL approach combined with AC is evaluated through a set of simulated experiments, verifying that an aggregate system eventually learns an improved scheduling policy reducing the overall system consumption. To this purpose, we adopt ScaFi [Cas+20a], which bundles, together with the language previously discussed, a simulator to execute aggregate programs.

Parameter	Description	Values
ϵ_0	ϵ at the beginning of simulations	[0.1, 0.4]
γ	Discount factor for Q update	[0.99, 0.95]
α	Learning rate for Q update	[0.1, 0.4]
w	State window stack size	[2, 5, 7]
θ	Balance of reactivity vs. consumption	[0.975, 0.9, 0.99]

Table 13.1: The parameters used in simulations. A simulation consist in a tuple of (ϵ_0, γ, α, w, θ).

For the sake of reproducibility, the code and the instructions to run simulations and produce the charts are open-sourced and available at a public repository[1].

13.3.1 Simulation Setup

In the experimental setup, we configure the network to consist of a 100x100 meter grid populated with $N = 100$ nodes. Each simulation episode is designed to last for 80 seconds of simulated time. For the training phase, we conduct $L = 1000$ episodes using an ϵ-greedy policy. The value of ϵ is dynamically adjusted in each episode k according to the formula:

$$\epsilon_k = \epsilon_0 - (\epsilon_0 / L \times k) \tag{13.3}$$

Here, ϵ_0 serves as the initial value of ϵ. This decay mechanism is implemented to strike an effective balance between exploitation and exploration as the training progresses.

In terms of learning, a single global Q-table is maintained, which each agent updates using local trajectory tuples of the form (s_t, a_t, r_{t+1}). Following the training phase, we evaluate the system's performance over the final $T = 50$ episodes, employing a greedy policy for this assessment. At this stage, each agent operates with a localized copy of the Q-table, reflecting the state of the system post-training.

The nodes in the grid have the option of executing four distinct actions, corresponding to different next-wake uptimes: 100ms, 200ms, 500ms, and 1s. The focus of the training will be on optimizing scheduling policies for two specific programs, referred to as the gradient-cast (G) and the converge-cast (C) building blocks, which are detailed in Part I.

To explore the impact of dynamic changes on the system, we introduce variability in the set of source nodes. When the source set changes, the computational field must recalibrate to a new stable state. Slow response from nodes can significantly extend the convergence time. In scenarios with minor changes, learning

[1] `https://github.com/cric96/experiment-2022-acsos-round-rl`

algorithms could favour solutions where nodes opt for the longest next wake-up time, thereby reducing responsiveness.

To induce this variability, we design two specific scenarios. In the first scenario, referred to as Swap, a single source node is initially placed at the grid's leftmost corner. At time 40, the rightmost node becomes the new source, causing a ripple effect in the system that is eventually stabilized by the aggregate computation.

The second scenario, termed MultiSwap, starts with a central node as the single source. At time 30, four nodes at the grid corners become new sources. At time 60, these corner nodes revert to being non-source nodes.

To assess the performance of our RL-based solution (Rl), we conduct simulations for each parameter set as shown in Table 13.1. We compare Rl against two alternatives: a fixed-rate scheduling approach (Periodic) with a minimum wake-up time of 100ms, and a power-minimizing heuristic (Ad-hoc).

Specifically, in Ad-hoc, nodes adjust their next wake-up time based on output changes. If the output differs from the previous step, they double the next wake-up time, up to a maximum limit. Otherwise, they keep it at the minimum action set value of 100ms.

We examine how different parameters influence learning dynamics and quantify the solution error using mean squared error (MRSE) between Periodic and Rl as follows:

$$MRSE_t = \sum_{i=0}^{N} (\text{output}_t(i)^{\text{Periodic}} - \text{output}_t(i)^{\text{RL}}) \tag{13.4}$$

In the above equation, t is the time step at which the error is evaluated and i indexes individual nodes. We also count the total number of computation rounds (or "ticks") across the entire system:

$$\text{TotalTicks}_t = \sum_{i=0}^{N} (\text{rounds}(i)) \tag{13.5}$$

Here, `rounds` is a local function that tallies how many times a node has executed the aggregate program. Another metric we consider is the rate of ticks per second:

$$\text{TicksPerSecond}_t = \text{TotalTicks}_t - \text{TotalTicks}_{t-1} \tag{13.6}$$

For a direct comparison between Rl and Ad-hoc, we calculate two additional metrics: the *error percentage* and the *energy saving percentage*. The error percentage indicates how much the computational field of Rl diverges from that of Periodic. The aim is to minimize this value. The energy-saving percentage is

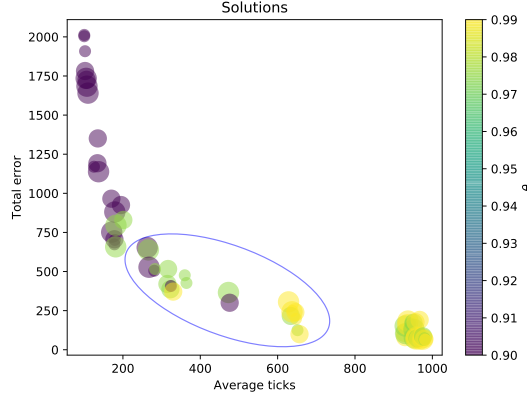


Figure 13.2: The distribution of the solutions after the training phases in SWAP scenario. Each point consists of one of the configurations expressed in Table 13.1. The colour of nodes shows the θ parameter. The size of nodes represents the w value (the smaller the node the smaller the w). The solutions of interest are located in the blue ellipse. Similar results are achieved with C and in the MULTISWAP scenario.

calculated as:

$$\text{EnergySaving}_t = \frac{\text{TotalTicks}_t^{\text{Periodic}} - \text{TotalTicks}_t}{\text{TotalTicks}_t^{\text{Periodic}}} \tag{13.7}$$

This metric is intended to be maximized, as a higher value indicates more significant power savings.

13.3.2 Discussion and Results

The simulations demonstrate that the learning algorithm produces different policies. Particularly, observing the Figure 13.2 we could recognize three macro-behaviours:

- the system tends to reduce the power consumption as much as possible, leading to high error and non-reactive policy. It happens mainly with a low value of θ since the smaller it is the greater the reward for reducing consumption tends to be.
- When the simulation runs with a high θ value, the policies remain with a high power profile to increment as much as possible the reactivity to the environment changes.

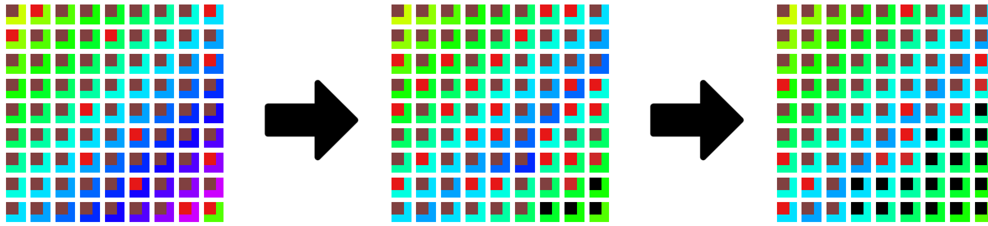


Figure 13.3: The slow-down behaviour of the policy learnt in SWAP scenario. The colours of the small squares denote the node frequencies (the redder, the higher the frequency). The colours of the large squares denote the output (the greener, the closer the nodes are to the sources). In the leftmost figure, a new source appears in the bottom right corner. The signal propagation produces a frequency drop on the node that evaluates the new value (the black node grows toward the gradient direction).

- The system tries to balance power efficiency and reactivity, which is our intended goal. It is typically achieved with a value in the middle of the two, but other parameters influence the training results.

This section discusses the results of those simulations that have a high power-saving percentage and a low error in each scenario/program since the others do not have any particular point of interest. Figure 13.4 shows the charts of the simulation results. In each scenario, the algorithm successfully reduces the power consumption percentage of the system, maintaining a low percentage of the output error. The best performances are typically reached with the highest value of θ, γ, and w. Indeed, the first value guides the learning process to reduce the period in which the node outputs are in an unstable condition. γ tends to reduce the long-term consumption, guiding the node to maximize the overall power. Finally, with a great value of w, agents tend to better understand the output progression and then they better react to local changes.

The best results are achieved in the SWAP (Figure 13.5) scenario and with the gradient program. Here, introducing at most a 10% of error in the swap moments (i.e., when the new source appears), the learning algorithm reduces the power consumption by nearly 60%. Particularly, the consumption is near to our AD-HOC solution, but with a remarkably reduced error. Interestingly, at the swap time, the round frequency drops until a peak and then soars (Figure 13.3). This behaviour may appear counter-intuitive. At a first glance, indeed, we expect that the nodes tend to maintain a low-cost power consumption, and then when changes in the environment happen, the nodes start to increase the frequency to react quickly against them. However, if the nodes sleep, they cannot intercept new events. For this reason, the agents tend to maintain a high power consumption to identify

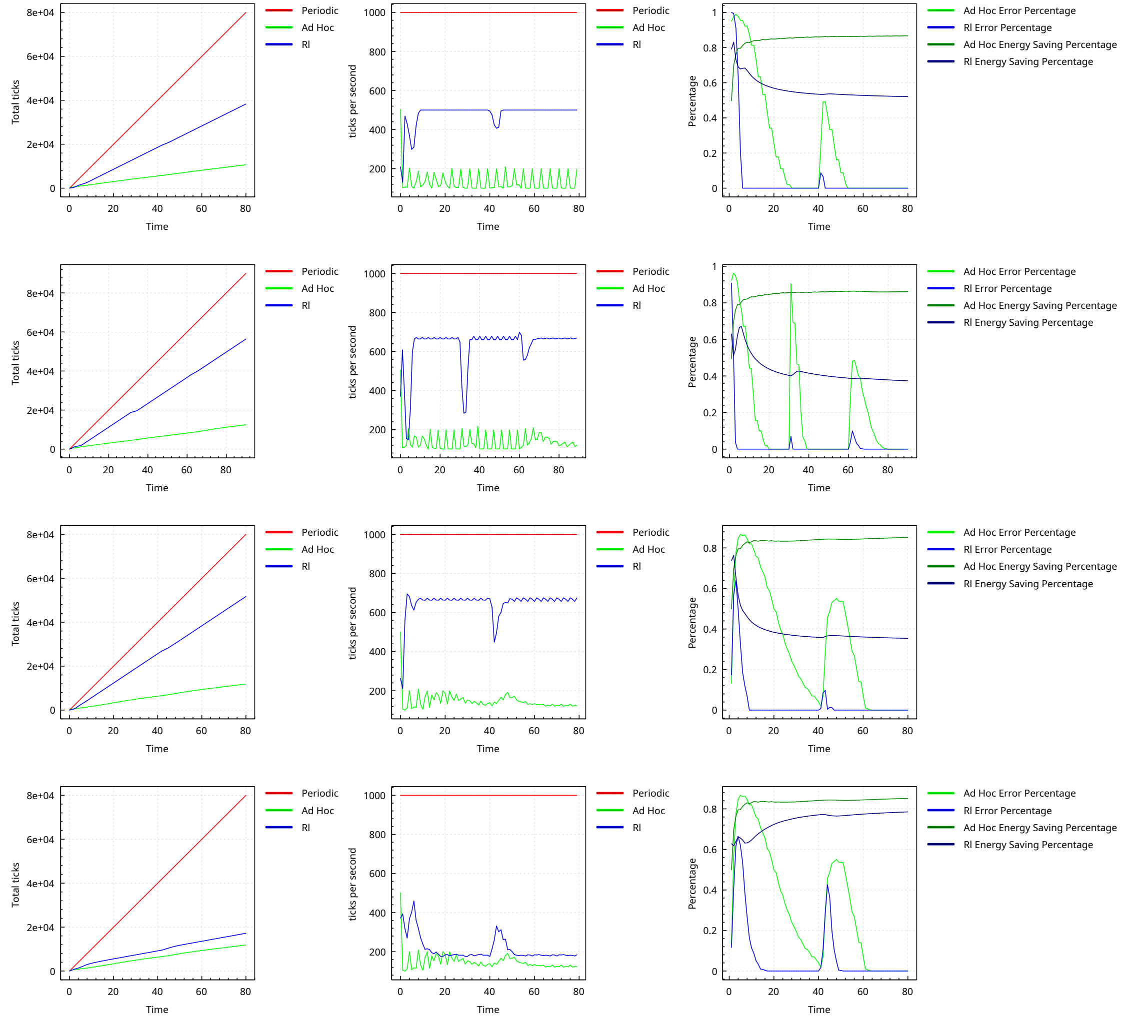


Figure 13.4: Simulation results. The leftmost chart shows the total ticks as time passes (*ticks*), the chart in the middle shows the ticks per second, and the rightmost chart shows the error and consumption measures. The first line shows the best performance of the gradient-case program in the simplest scenario (i.e., SWAP). The second line evaluates the performance of the gradient in the MULTISWAP scenario. Finally, the last two lines show how RL manages the C block, with different θ values. The overall power-saving using our approach is between 60% to 40% with respect to the PERIODIC program evaluation.

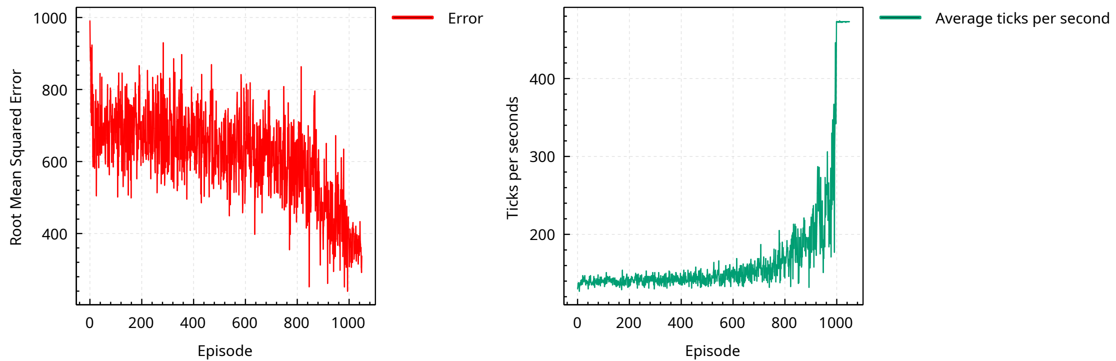


Figure 13.5: Shows the average error and the average ticks during the learning episodes of SWAP scenario.

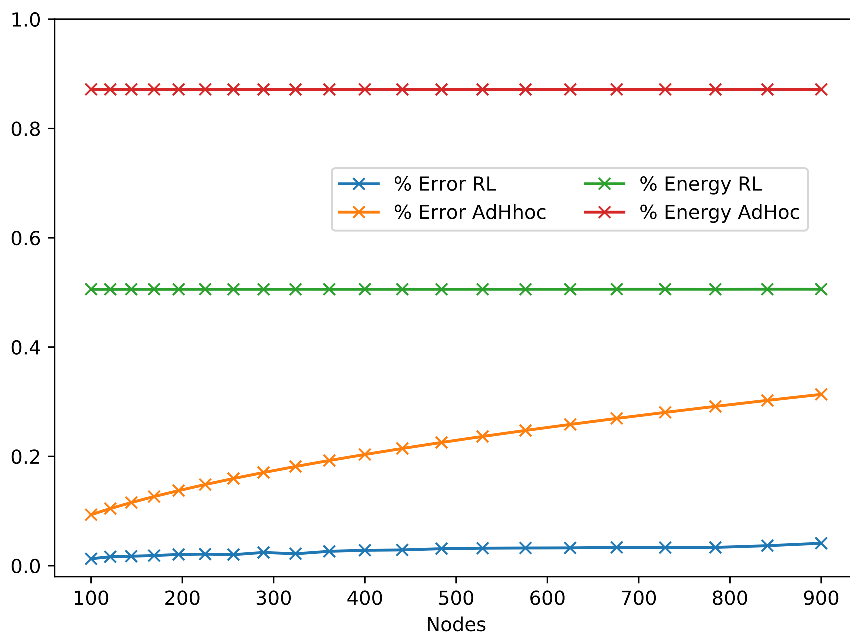


Figure 13.6: Error and energy saving percentage (see Section 13.3.1) as nodes vary. We use the best policy found in the SWAP scenario as a reference, checking how the error and energy-saving change.

changes, and then they could enter power-saving settings.

In the MultiSwap scenario, the algorithm reaches a good performance with a very low error, but with slightly higher power consumption (600 ticks on average versus 500 of the Swap scenario). This happens because we gave more importance to the convergence time (with $\theta = 0.99$) and therefore the algorithm tended to prefer solutions with low error. Nevertheless, even if it has higher consumption, it reaches a similar performance of Periodic but with half of the ticks.

The same RL settings could be used even with different programs. Indeed, the system learns how to reduce consumption when it is used with block C. Interestingly, θ could be used to decide the trade-off between power consumption and reactivity: the program with $\theta = 0.9$ (fourth line Figure 13.4) has a low error but a limited reduction in the power consumption (~ 40 %); instead, the program with $\theta = 0.975$ has an increased convergence time in the swap moments but the overall consumption is reduced by a factor of 70.

Lastly, we want to recall that AC programs are scale-free regarding the number of nodes, since they leverage self-organization to reach a collective structure. Therefore, even the learnt policy should not depend on the agents in the system. To verify this consideration, we use the same policy found with 100 nodes in several other deployments (from 100 nodes to 900). Particularly, in Figure 13.6, the power consumption reduction remains stable as the nodes vary. Moreover, the error is constant too. There were some oscillations but the error remains negligible even if the size is 10 times the nodes of the training configuration. Differently, the heuristic worsens by a factor of three.

13.3.3 On practical applicability

We test our algorithm in simulated scenarios, but it can simply be adapted in a real system, which mainly means:

1. define the training phase (offline/online);
2. integrate the RL agent inference to the aggregate middleware

For 1) if the learning is performed online, it should be also taken into account the cost of the central server that performs the learning and the communication among the nodes— or any technique that allows a global Q table to be maintained and consistently updated (e.g., via gossip algorithm). Obviously, in the case of very dense and large-scale networks that is significant. Whereas if simulations are used, the cost of learning is negligible. Currently, our focus has been on the second case—implementing true distributed online learning, a more detailed evaluation is needed to understand the cost of maintaining a central and updated Q table. For 2) on the other hand, there are several aspects to consider. In fact, AC can adapt to

different computing platforms and communication protocols, as it adopts a fluid approach—see work on pulverization [Cas+20b]. Thus, although AC typically does not care about the underlying platform, since our work is heavily dependent on the execution model instead, we have to be sure that this is closer to reality. Practically, we have to check i) the communication model and ii) the scheduling platform. For ii), the devices should be able to change the energy-saving model at runtime to support our AC solution—that is already possible in various embedded systems such as ESP-32. Regarding communication instead, it could be that some scheduling policies that rely on a communication protocol do not work for another protocol (e.g., a policy trained with wired TCP, does not work well with wireless UDP due to packet collisions, message flooding, etc). Therefore, in the case of simulation, we should have a communication model as detailed as possible to be sure that the policy works in the selected platform too.

13.4 Final Remarks

This chapter explores the combination of in combining RL with AC to optimize non-functional aspects of collective computations such as power consumption, energy bandwidth, and reactivity (in terms of convergence time) as highlighted by the research roadmap. In particular, this work leverages Q-Learning to reduce the cost of executing AC programs in several synthetic scenarios. The contribution can be framed within a larger vision in which the designer could mainly focus on functionality and key non-functional aspects (e.g., resiliency) at design time, while the platform is programmed to or instructed to learn how to optimize less critical but still highly desired non-functional aspects—acting upon scheduling or deployment [Cas+20b].

Chapter 14

Platform: Toolkit for Hybrid Aggregate Computing

How can we design a framework for the development of effective ManyRL systems
– **RQ2, RQ3, RQ4**

Contents

14.1 Software Description 254
14.1.1 Core abstraction 255
14.1.2 ScaFi-Alchemist integration 257
14.1.3 DSL for learning configurations 258
14.1.4 Tool usage 260
14.2 Experiments 260
14.2.1 Description 260
14.2.2 Results 261
14.3 Related work 262
14.3.1 Many Agent simulators: 263
14.3.2 Multi-Agent Deep RL libraries: 263
14.4 Final Remarks 265

In this dissertation, we advocate for a language-based approach to the engineering of CPSW. Specifically, we propose a hybrid vision that combines RL and aggregate computing. This approach underscores the need to create a corpus of solutions

to address the overarching problem of ManyRL. ManyRL [Yan21; HDB21] offers significant opportunities in the design of large-scale systems requiring agents to coordinate and collaborate effectively, even in environments with partial observability and intrinsic unpredictability. Therefore, there is a need for effective frameworks (and tools) that can foster ManyRL adoption.

However, although several frameworks exist both for describing and solving ManyRL problems (Ray [Mor+17]) and for using and defining multi-agent environments (PettingZoo [Ter+21]) they generally lack the following aspects: *(i)* setting up complex environments (and hence simulation scenarios) is particularly difficult, and *(ii)* they are typically tailored for handling a limited number of agents, and for non-collaborative tasks.

To start addressing these problems, in this work, we present ScaRLib, a framework for the design of effective ManyRL systems, that provides the following set of features: support for centralized training and decentralized execution, easy extensibility, a DSL for easily expressing complex cooperative scenarios, integration with a simulator for large-scale pervasive computing systems (Alchemist), and the possibility to express field-based coordination problems thanks to the integration with ScaFi. With ScaFi, in particular, we support our language-based vision: ScaRLib is provided with a high-level language for distributed computing that provides declarative and compositional ways of expressing complex coordination tasks.

14.1 Software Description

ScaRLib [1] [2] is a research Scala framework designed to support the development of ManyRL systems by JVM-based high-level specification, and with learning performed under the hood by PyTorch[3]. This project aims to provide a tool that allows easy and powerful system specification. To meet this purpose we have designed many abstractions, that model high-level aspects of the ManyRL domain, without caring about low-level implementation details. Basically, ScaRLib is composed of three main modules (Figure 14.1), namely:

- `scarlib-core` that implements the main abstractions over the ManyRL domain,
- `dsl-core` that provides a high-level language to specify the system, and

[1] Tool available on GitHub at `https://github.com/ScaRLib-group/ScaRLib`
[2] demo video at: `https://github.com/ScaRLib-group/ScaRLib-demo-video`
[3] `https://pytorch.org/`

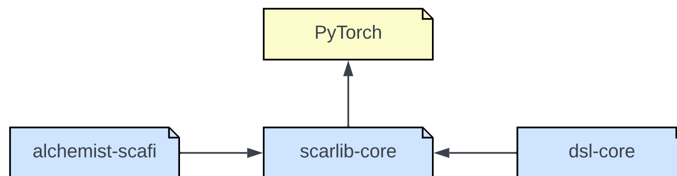


Figure 14.1: ScaRLib main modules

- `alchemist-scafi` that provides bindings between ScaRLib and the two tools Alchemist and ScaFi. It is important to note that ScaRLib is not limited to the Alchemist-ScaFi combination, that module is already implemented due to the actual need, however, it is possible to implement other bindings to other tools (e.g., by replacing Alchemist with some other simulator, for example, FLAME GPU [RCR09]).

14.1.1 Core abstraction

The module `scarlib-core` implements the core functionalities and abstractions of the framework, such as the definition of the main data structures and the implementation of the main algorithms. All the abstractions (Figure 14.2) are built around a bunch of concepts. The key element is the `System`, which is a collection of agents that interact within a shared environment and that are trained to optimize a global or local reward signal expressed by a reward function. The tool comes with two types of systems already implemented that are very common in literature [DD20], i.e., centralized training and decentralized execution (`CTDESystem`) and decentralized training and execution (`DTDESystem`). Furthermore, an implementation of the DQL algorithm [Mni+15] is provided and used to train agents. The end-user who wants to run a learning process only has to implement four elements in order to define his own system with the desired *collective* goal, which are:

1. the *environment*: that is the place where the agents live,
2. the *agent state space*: namely the information that the agent can perceive from the environment,
3. the *action space*: namely the actions that the agent can perform in that environment, and
4. the *reward function*: that is the function that the agent has to maximize.

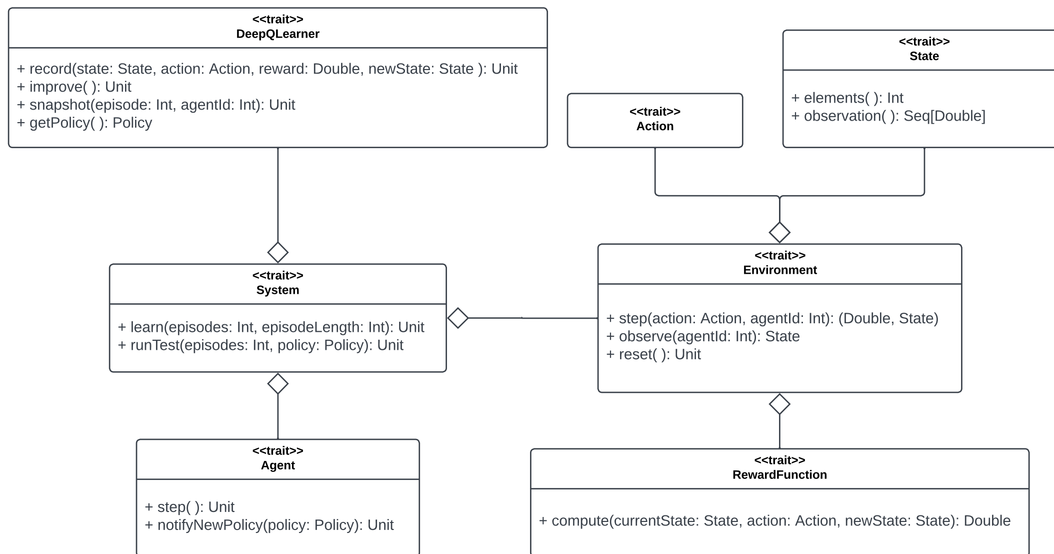


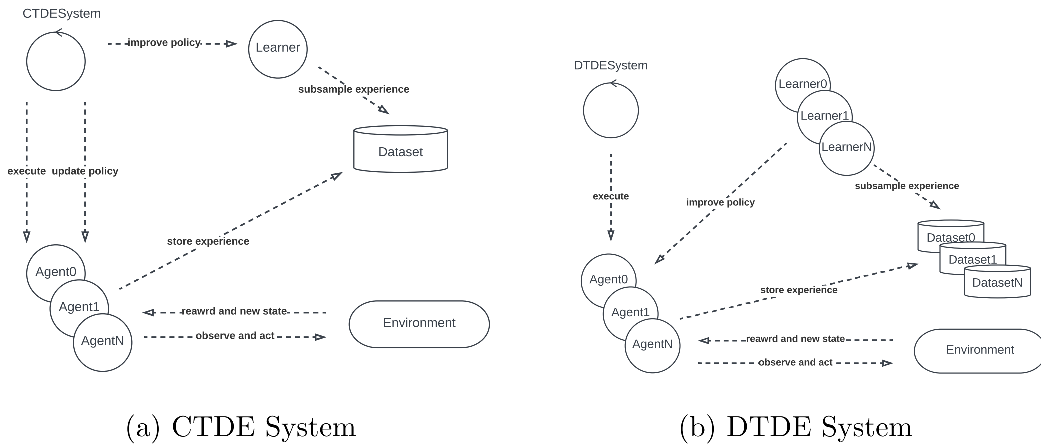
Figure 14.2: ScaRLib core architecture

Only by using this module, it is possible to run a simple learning process in a simulated environment based on our platform.

To gain a deeper comprehension of the system dynamics, a discussion of the underlying mechanisms is warranted. Both systems employ a training algorithm characterized by a sequence of *epochs*, each encompassing a collection of *episodes*. Within each episode, agents are presented with the current state, which serves as an input for action selection. The collective action executed by the agents prompts a state transition in the environment, thereby facilitating the progression to the subsequent episode. Upon completion of an epoch, the environment is reinitialized, and the agents undergo training based on the aggregated experience.

Most specifically, if the chosen system is a `CTDESystem` (Figure 14.3a) the agents are trained in a centralized way, for that reason, there is a single central dataset, where the global experience of all the agents is stored, and a single central learner that is responsible for the training process and for the improvement of the policy neural network. The system is also responsible for the execution of all the agents and the notification of the updated policy. In this way, it is possible to easily extend the system in order to modify the execution flow, e.g., if a concurrent and distributed execution is needed. The `DTDESystem` (Figure 14.3b) works similarly, the only difference is that every agent has its own dataset and learner.

Regarding the training process, since the tool aims to support neural-network-based RL algorithms (like DQN), we chose to use the current de facto standard framework for building neural networks, which is PyTorch—alternatives include



(a) CTDE System (b) DTDE System

Figure 14.3: Examples of developed system dynamics. On the left, there is the centralized system, where a learner with a global view of the system updates the policy shared with all agents. On the right, there is a decentralized system, where each agent has a local policy and a local policy.

DL4J [4], which could be subject of future investigation.

One way to integrate this library into a JVM environment could be to rely on its native core (LibTorch) using Java Native Interface (JNI) – as was done in `scala_torch` [5] project. In ScaRLib, we chose a convenient approach that allowed us not only to access PyTorch but also all the libraries connected to it (e.g., torch geometric, etc.), which is to use ScalaPy [LS20] to interact directly with the Python API of these libraries. This integration generally involves:

1. setting up a Python environment in which the libraries of interest are instantiated;
2. creating a Scala API that isolates what is necessary to access the Python ecosystem.

In this case, we have isolated everything in DQN, which is therefore the entry point for accessing PyTorch.

14.1.2 ScaFi-Alchemist integration

In addition to the core, we have implemented another module called `alchemist-scafi` (Figure 14.4) in which there is the integration with the two

[4] `https://deeplearning4j.konduit.ai/`
[5] `https://github.com/microsoft/scala_torch`

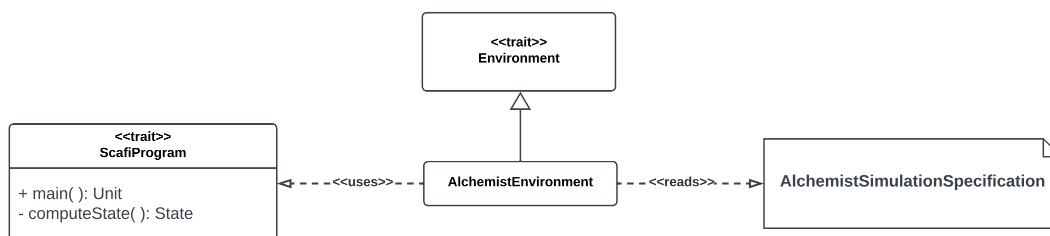


Figure 14.4: ScaRLib `alchemist-scafi` architecture. A `ScafiProgram` should be passed to the `AlchemistEnvironment` in order to start the learning process.

tools: Alchemist and ScaFi. Such integration enables the possibility to run the learning process in an aggregate computing context. This is a key part of this contribution. In fact, although Alchemist has been used for cooperative many-agent reinforcement learning with ScaFi as shown in previous chapters, ad-hoc solutions were always created that were difficult to *reuse*, *rigid*, *untested*, and had *interoperability issues* between Alchemist and the chosen native libraries. With this integration, we aim to deliver a robust and user-friendly system that will serve as a long-term solution, fostering greater engagement within the multi-agent reinforcement learning community. This simulator and paradigm have already demonstrated their versatility in representing a wide range of environments, as discussed in Part I.

The specification of a learning system does not change, only two new elements are added: the specification of the Alchemist simulation and the implementation of the ScaFi-based logic. Specifically, the Alchemist simulation is configured as depicted in Figure 3.9, where a ScaFi class, encapsulating the aggregate programming code, is passed as a program. To facilitate the training progression, a molecule containing the current action–a subclass of the `Action` class–is integrated within the ScaFi program. This molecule is injected by a learner governing the RL policy. Additionally, the aggregate program evaluates the environment state, which is required to be a subtype of the `State` class, via the `computeState` method. This computed state is subsequently encapsulated in the `state` molecule, serving as input for the learner to refine the policy.

14.1.3 DSL for learning configurations

Finally, we developed an internal DSL designed to streamline the development of ManyRL training systems. This approach aims to seamlessly translate the concepts envisioned by a ManyRL system designer into a functional training system. Utilizing a strongly-typed language like Scala for our DSL enables error detec-

tion at compile-time, thereby preempting simple configuration errors that might otherwise only be caught during runtime.

The public-facing DSL serves as a simplified interface to the underlying abstractions encapsulated in the `scarlib-core` module. Consequently, when a developer wishes to initiate a simulation (for example, in Alchemist), they must first specify a reward function. This function serves as a metric for evaluating the performance of a particular agent in relation to the current state of the environment.

```
class MyRewardFunction extends CollectiveRewardFunction:
    override def computeReward(
        state: State,
        action: Action,
        nextState: State
    ): Double = ...
```

Consequently, they must decide which actions are supported by the agents living in the chosen system. Since we are talking about ManyRL systems, we suppose that each agent has the same action space. Thus, it is possible to define a set of actions as a product type:

```
sealed trait MyAction extends Action
object MyAction:
    case object A extends MyAction
    case object B extends MyAction
    case object C extends MyAction
    def all: Seq[MyAction] = Seq(A, B, C)
```

Final refinements required include:

- choosing the class of the Alchemist environment to instantiate,
- defining the number of agents living in the chosen environment, and
- defining the size of the buffer in which the memory will be stored.

This can be expressed as follows:

```
val system = learningSystem {
    rewardFunction { new MyRewardFunction() }
    actions { MyAction.all}
    dataset { ReplayBuffer[State, Action](10000) }
    agents { 50 } // select the number of agent
    environment {
        // select a specific environment
        "it.unibo.scarlib.experiments.myEnvironment"
    }
}
```

14.1.4 Tool usage

The tool is published on Maven Central and it is possible to include it in your project, for example, through a build system. In the case of Gradle, for instance, you will need to add the following instructions:

```
implementation("io.github.davidedomini:scarlib-core:1.5.0")
implementation("io.github.davidedomini:dsl-core:1.5.0")
```

At this point, it will be possible to create your own training system as shown in the DSL section. To start the training, you will then need to write:

```
learningSystem.train(episodes = 1000, episodeLength = 100)
```

Of course, the system can also be used to verify a certain policy that has been learned during a training process. To do this, first, you will need to load the neural network extracted during training:

```
val network = PolicyNN(path, inputSize = ..., hiddenSize = ...)
```

Then you can execute the test in the following way:

```
system.runTest(episodeLength = 100, network)
```

For further details on how to specify simulations and environments, please refer to the repository README, the presentation video and the developed simulation (following section).

14.2 Experiments

14.2.1 Description

To test ScarLib's functionality, we develop an experiment [6] involving a relatively large number of agents and non-trivial coordination tasks. We aim to create a flock of drones that moves to avoid collisions with each others, by learning a policy by which each agents decide how to move based on neighbours relative position. This is a well-known problem, and various models and algorithms exist which we draw upon [Rey87; Šoš+16]. In this case, we assumed that agents position themselves in an unlimited 2D environment with a fixed neighbourhood (the closest five, in our experiments, though this is a simulation parameter) and have the ability to perform movement steps in the 8 directions of a square grid (horizontally, vertically, or diagonally). The environment state, as perceived by the single agent, is the relative distance to the closest neighbours. Particularly, it was expressed through ScaFi as:

[6] repository available at https://github.com/ScaRLib-group/ScaRLib-flock-demo

```
val state = foldhoodPlus(Seq.empty)(_ ++ _)(Set(nbrVector))
```

where `nbrVector` is the vector representing the relative position of the neighbour. `foldhoodPlus` is a ScaFi function that allows to fold over the neighbourhood and `++` is the concatenation operator for sequences.

The crucial point for this task is the definition of the reward function. In this simulation, we based it on *collision* and *cohesion* factors. We aim to learn a policy by which agents, initially spread in a very sparse way in the environment, move toward each other until reaching approximate δ distance without colliding, ultimately forming one or many close groups.

The collision factor comes into play when the distance is less than δ, and exponentially weighs the distance d relative to its closest neighbour:

$$\text{collision} = \begin{cases} 0 & \text{if } d > \delta \\ \exp\left(-\frac{d}{\delta}\right) & \text{otherwise} \end{cases} \tag{14.1}$$

In this way, when the negative factor is taken into account: the system will tend to move nodes away from each other.

However, if only this factor were used, the system would be disorganized. This is where the cohesion factor comes in. Given the neighbour with the maximum distance D, it linearly adjusts the distance relative to the node being evaluated by function:

$$\text{cohesion} = \begin{cases} 0 & \text{if } d < \delta \\ -(D - \delta) & \text{otherwise} \end{cases} \tag{14.2}$$

The overall reward function is defined as the sum of these two factors ($cohesion + collision$) as shown in Figure 14.5.

14.2.2 Results

To verify the functionality of the described simulation, we divided the evaluation into two parts. In the first part, we trained the system for a total of 1000 epochs, each consisting of 100 episodes (or steps). For each epoch, we randomly place 50 agents in a grid large 50x50 meters. We set the target distance δ at 2 meters.

Given the flexibility of ScaRLib, we tested the training with both CDTE and DTDE processes to ensure that the system could produce policies capable of solving the described task in both cases. With the homogeneous policies found (i.e., the one extracted from the CTDE process), we verified that the system's behaviour was consistent with what was learned by varying the initial seed in 16 simulations. With the CDTE policy, since we considered the system homogeneous, we also verified the behaviour as the number of nodes varied, expecting similar performance as the nodes increased.

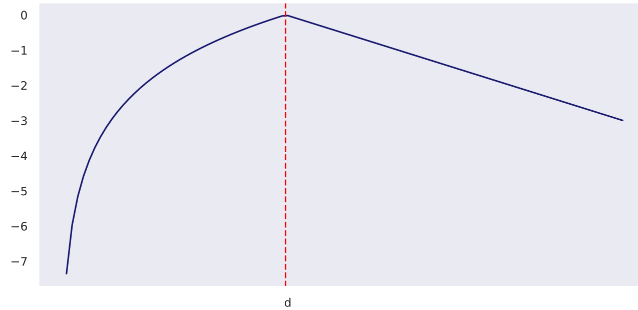


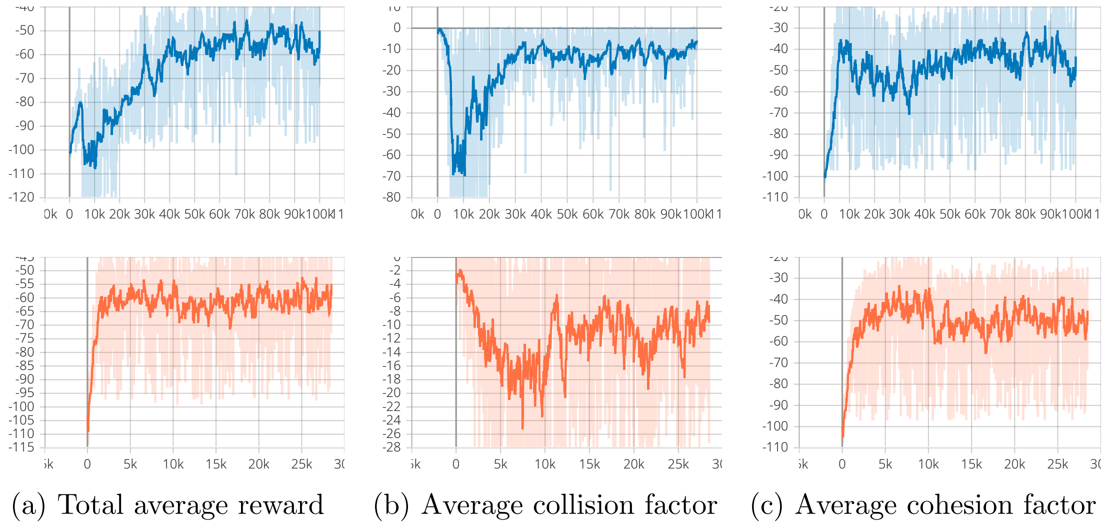
Figure 14.5: Cohesion-Collision reward function: the red vertical line represents the target distance d. The portion of the graph to the right of the red line represents the influence of the cohesion term, while the left one represents the influence of the collision term.

The graphs shown in Figure 14.6 demonstrate the *multi-objective* nature of the problem. In fact, cohesion and collision are two contrasting signals, and the system had to find a balance between these two values. The graphs show that DQN can generally optimize one signal at a time, with cohesion tending towards zero and collision increasing. Nonetheless, after 500 epochs in CTDE simulation, we see that the system had already found a balance between these two factors. In the case of DTDE learning, we observe that convergence is achieved in fewer steps ($\tilde{5}0$). This is because there is a greater number of policies and therefore greater overall complexity compared to a single homogeneous policy.

During the testing phase (Figure 14.7 shows a series of snapshots of the learned policy), we observed that the system is capable of maintaining a distance of approximately δ, both in the CDTE and DTDE cases. Most specifically, we note that the homogeneous policy is generally a winning choice for homogeneous ManyRL tasks. Increasing the number of agents (from 50 to 200), we can observe that collective performances are similar to those with few agents (Figure 14.8).

14.3 Related work

MARL has gained significant interest in the past decade, leading to the development of several frameworks for use in both research and industry communities. Here, we highlight current state-of-the-art solutions for MARL problems and compare them to the tools presented in this work.



(a) Total average reward (b) Average collision factor (c) Average cohesion factor

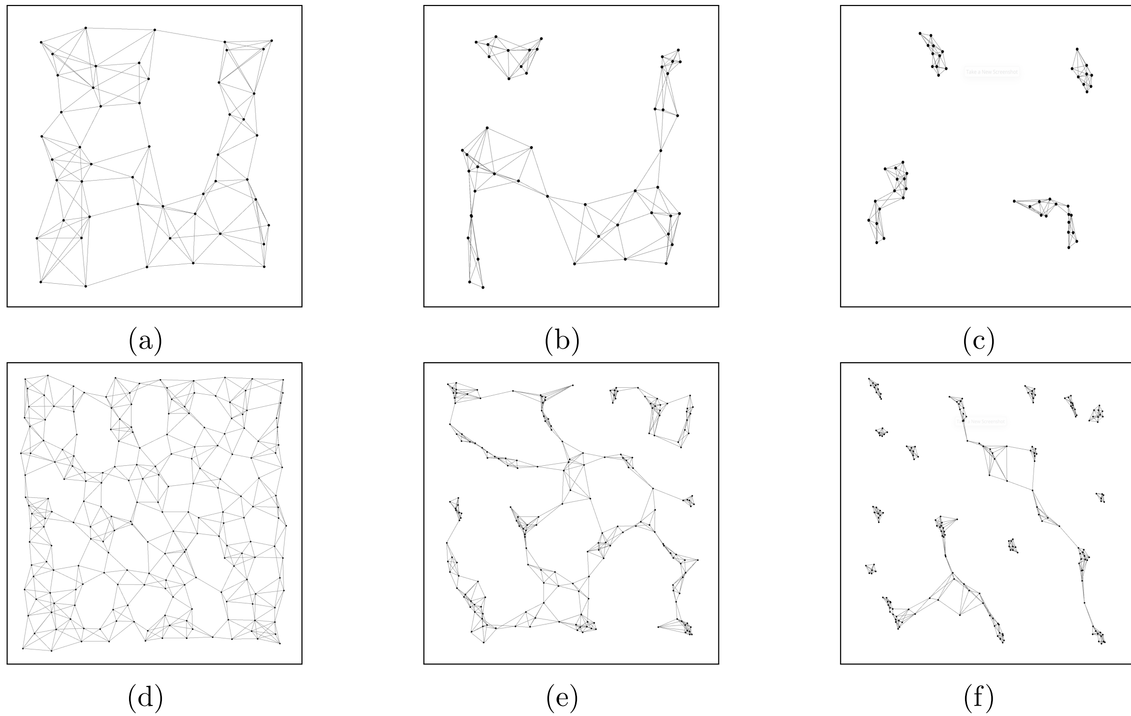
Figure 14.6: Cohesion and collision experiment results. The y-axis represents the reward value. The x-axis represents the total number of episodes. The first three graphs show the results of the CDTE learning process, while the last three show the results of the DTDE learning process.

14.3.1 Many Agent simulators:

Unlike supervised learning, where a large dataset is required to improve neural network performance, in RL, algorithms require a simulator to gain experience. One such comprehensive solution for MARL is PettingZoo [Ter+21], which provides both competitive and cooperative settings for simulations with multiple agents. Another option for many-agent scenarios is NeuralMMO [Sua+19], a GPU-optimized simulator for MMO-like games that is designed to handle large-scale simulations of thousands of agents. Vectorized Multi-agent Simulator [Bet+22] is another promising solution, as it is optimized for collective tasks through GPU computation, and it can be extended with additional environments. While ScaR-Libis not directly linked to any simulator, its main abstraction can be potentially linked to both JVM-based simulators and gym-based Python environments. Our choice of Alchemist was mainly due to its ability to express ManyRL settings easily, but potentially it can be used with any of the above-described solutions.

14.3.2 Multi-Agent Deep RL libraries:

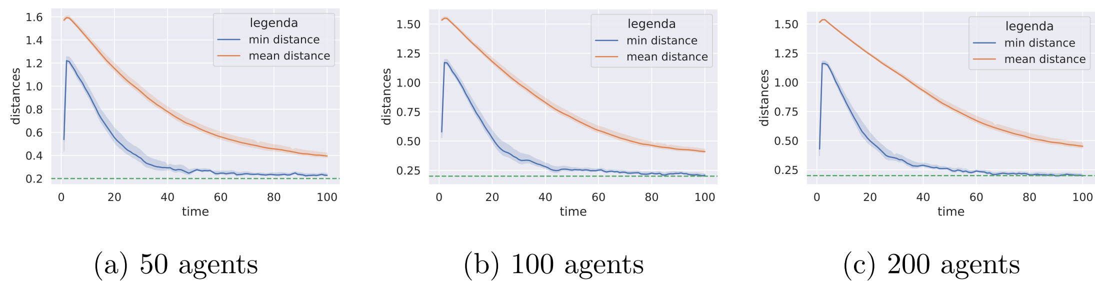
since the importance of multi-agent settings several libraries have been developed in recent years. Ray [Mor+17] is one of the most comprehensive frameworks, originally designed for single-agent RL but now integrated with basic concepts for MARL solutions thanks to MALib. It offers various MARL algorithms, supports



(a) (b) (c)

(d) (e) (f)

Figure 14.7: Snapshots of the learned policy, the time flow is from left to right. In the first row, there are 50 agents, whereas in the second row, there are 200 agents. In the last step of the simulation, the agents converged to a distance of approximately δ.



(a) 50 agents (b) 100 agents (c) 200 agents

Figure 14.8: The performance of the learned policy. The y-axis represents the distance between the agents. The x-axis represents the time. The green line is equal to δ. In the charts, as the number of agents varies, the performance of the learned policy is similar. Moreover, the minimum (blue line) distance between the agents is always greater than δ. The average distance (orange line) stays close to $2 * \delta$ (after convergence).

different gym-like environments, and is highly customizable through configuration files. PyMARL [Sam+19] is one of the first solutions in Python for MARL, though it is limited to specific algorithms (like VDN and QMIX), and it is not generalizable. ScaRLib is more similar to the first framework, even though it is primarily designed for cooperative applications. However, since it was developed specifically for ManyRL, it includes some abstractions and configurations that are not present in Ray, such as the concept of a collective reward function and the configuration for CTDE. This reduces the time required to use ScaRLib compared to Ray. Additionally, ScaRLib has a simple DSL that is easier to use than Ray's configuration system and is aided by the type system.

Finally, some innovative approaches aim to scale solutions to large populations of cooperative agents, such as mean-field RL. However, only a few implementations currently exist, and they are not considered to be general-purpose. ScaRLib, on the other hand, offers a practical, simple implementation that can be leveraged in ManyRL settings.

Overall, ManyRL is a high-level framework that reduces the effort required for developers and practitioners to define and implement ManyRL problems when compared to current state-of-the-art solutions.

14.4 Final Remarks

In this chapter, we presented ScaRLib: a collaborative many-agent deep reinforcement learning framework that integrates the functionalities of ScaFi and Alchemist. The framework enables the definition of simulations of large-scale distributed scenarios and the creation of complex scenarios with ease through its exposed DSL. With ScaRLib, developers can effectively and efficiently simulate and experiment with different reinforcement learning algorithms, thereby providing a valuable tool for the advancement of coordination and multi-agent systems research. This tool is essential in the view presented in this thesis: the development of Cyber-Physical Swarms applications need both a foundational theory and a practical tool to test and validate the theory.

References

[Agu21] Gianluca Aguzzi. "Research directions for Aggregate Computing with Machine Learning". In: *Proc. of the Int. Conf. on Autonomic Computing and Self-Organizing Systems*. IEEE, 2021, pp. 310–312. DOI: 10.1109/ACSOS-C52956.2021.00078.

[AT04] Adrian K. Agogino and Kagan Tumer. "Unifying Temporal and Structural Credit Assignment Problems". In: *3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), 19-23 August 2004, New York, NY, USA*. IEEE Computer Society, 2004, pp. 980–987. DOI: 10.1109/AAMAS.2004.10098. URL: http://doi.ieeecomputersociety.org/10.1109/AAMAS.2004.10098.

[Aud+17] Giorgio Audrito et al. "Compositional Blocks for Optimal Self-Healing Gradients". In: *11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017, Tucson, AZ, USA, September 18-22, 2017*. IEEE Computer Society, 2017, pp. 91–100. DOI: 10.1109/SASO.2017.18. URL: http://doi.ieeecomputersociety.org/10.1109/SASO.2017.18.

[Aud+21] Giorgio Audrito et al. "Optimal resilient distributed data collection in mobile edge environments". In: *Comput. Electr. Eng.* 96.Part (2021), p. 107580. DOI: 10.1016/j.compeleceng.2021.107580. URL: https://doi.org/10.1016/j.compeleceng.2021.107580.

[Bea+08] Jacob Beal et al. "Fast self-healing gradients". In: *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC)*. ACM, 2008, pp. 1969–1975. DOI: 10.1145/1363686.1364163.

[Bea+13] Jacob Beal et al. "Organizing the aggregate: Languages for spatial computing". In: *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, 2013, pp. 436–501.

[Bet+22] Matteo Bettini et al. "VMAS: A Vectorized Multi-Agent Simulator for Collective Robot Learning". In: *The 16th International Symposium on Distributed Autonomous Robotic Systems* (2022).

[BZI00] Daniel S. Bernstein, Shlomo Zilberstein, and Neil Immerman. "The Complexity of Decentralized Control of Markov Decision Processes". In: *Proc. of the Conf. in Uncertainty in Artificial Intelligence*. 2000, pp. 32–37.

[Cas+19] Roberto Casadei et al. "Self-organising Coordination Regions: A Pattern for Edge Computing". In: *Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Proceedings*. Vol. 11533. Lecture Notes in Computer Science. Springer, 2019, pp. 182–199. DOI: 10.1007/978-3-030-22397-7_11. URL: https://doi.org/10.1007/978-3-030-22397-7%5C_11.

[Cas+20a] Roberto Casadei et al. "FScaFi : A Core Calculus for Collective Adaptive Systems Programming". In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12477. Lecture Notes in Computer Science. Springer, 2020, pp. 344–360. DOI: 10.1007/978-3-030-61470-6_21. URL: https://doi.org/10.1007/978-3-030-61470-6%5C_21.

[Cas+20b] Roberto Casadei et al. "Pulverization in Cyber-Physical Systems: Engineering the Self-Organizing Logic Separated from Deployment". In: *Future Internet* 12.11 (2020), p. 203. DOI: 10.3390/fi12110203. URL: https://doi.org/10.3390/fi12110203.

[Cas+21] Roberto Casadei et al. "Engineering collective intelligence at the edge with aggregate processes". In: *Eng. Appl. Artif. Intell.* 97 (2021), p. 104081. DOI: 10.1016/j.engappai.2020.104081. URL: https://doi.org/10.1016/j.engappai.2020.104081.

[Cas+22a] Roberto Casadei et al. "Digital Twins, Virtual Devices, and Augmentations for Self-Organising Cyber-Physical Collectives". In: *Applied Sciences* 12.1 (2022). ISSN: 2076-3417. DOI: 10.3390/app12010349. URL: https://www.mdpi.com/2076-3417/12/1/349.

[Cas+22b] Roberto Casadei et al. "ScaFi: A Scala DSL and Toolkit for Aggregate Programming". In: *SoftwareX* 20 (2022), p. 101248.

[Cas22] Roberto Casadei. "Macroprogramming: Concepts, State of the Art, and Opportunities of Macroscopic Behaviour Modelling". In: *CoRR* abs/2201.03473 (2022). arXiv: 2201.03473. URL: https://arxiv.org/abs/2201.03473.

[CAV21] Roberto Casadei, Gianluca Aguzzi, and Mirko Viroli. “A Programming Approach to Collective Autonomy”. In: *J. Sens. Actuator Networks* 10.2 (2021), p. 27. DOI: 10.3390/jsan10020027.

[CPT10] Cristiano Castelfranchi, Giovanni Pezzulo, and Luca Tummolini. “Behavioral Implicit Communication (BIC): Communicating with Smart Environments”. In: *Int. J. Ambient Comput. Intell.* 2.1 (2010), pp. 1–12. DOI: 10.4018/jaci.2010010101.

[DD20] Wei Du and Shifei Ding. “A survey on multi-agent deep reinforcement learning: from the perspective of challenges and applications”. In: *Artificial Intelligence Review* 54.5 (Nov. 2020), pp. 3215–3238. DOI: 10.1007/s10462-020-09938-y. URL: https://doi.org/10.1007/s10462-020-09938-y.

[FL19] Matthias Fey and Jan Eric Lenssen. “Fast Graph Representation Learning with PyTorch Geometric”. In: *CoRR* abs/1903.02428 (2019). arXiv: 1903.02428.

[Gil+17] Justin Gilmer et al. “Neural Message Passing for Quantum Chemistry”. In: *Proc. of the 34th International Conference on Machine Learning.* 2017, pp. 1263–1272.

[Gos+22] Walker Gosrich et al. “Coverage Control in Multi-Robot Systems via Graph Neural Networks”. In: *Proc. of the Int. Conf. on Robotics and Automation.* IEEE, 2022, pp. 8787–8793. DOI: 10.1109/ICRA46639.2022.9811854.

[GPS17] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. “Program Synthesis”. In: *Found. Trends Program. Lang.* 4.1-2 (2017), pp. 1–119. DOI: 10.1561/2500000010. URL: https://doi.org/10.1561/2500000010.

[HCL18] Thomas Hendriks, Miguel Camelo, and Steven Latré. “Q^2-Routing : A Qos-aware Q-Routing algorithm for Wireless Ad Hoc Networks”. In: *14th International Conference on Wireless and Mobile Computing, Networking and Communications, WiMob 2018, Limassol, Cyprus, October 15-17, 2018.* IEEE, 2018, pp. 108–115. DOI: 10.1109/WiMOB.2018.8589161.

[HDB21] Keyang He, Prashant Doshi, and Bikramjit Banerjee. *Many Agent Reinforcement Learning Under Partial Observability.* 2021. DOI: 10.48550/ARXIV.2106.09825. URL: https://arxiv.org/abs/2106.09825.

[HKT19] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. "A survey and critique of multiagent deep reinforcement learning". In: *Auton. Agents Multi Agent Syst.* 33.6 (2019), pp. 750–797. DOI: 10.1007/s10458-019-09421-1. URL: https://doi.org/10.1007/s10458-019-09421-1.

[Iwa+21] Takuya Iwaki et al. "Multi-hop sensor network scheduling for optimal remote estimation". In: *Autom.* 127 (2021), p. 109498. DOI: 10.1016/j.automatica.2021.109498.

[KC03] Jeffrey O. Kephart and David M. Chess. "The Vision of Autonomic Computing". In: *Computer* 36.1 (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055. URL: https://doi.org/10.1109/MC.2003.1160055.

[KO22] Dongkwan Kim and Alice Oh. "How to Find Your Friendly Neighborhood: Graph Attention Design with Self-Supervision". In: *CoRR* abs/2204.04879 (2022). DOI: 10.48550/arXiv.2204.04879. arXiv: 2204.04879.

[KTA19] Boris Knyazev, Graham W. Taylor, and Mohamed R. Amer. "Understanding Attention and Generalization in Graph Neural Networks". In: *Conf. on Advances in Neural Information Processing Systems.* 2019, pp. 4204–4214.

[Le+19] Duc Van Le et al. "Control of Air Free-Cooled Data Centers in Tropics via Deep Reinforcement Learning". In: *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation, BuildSys 2019, New York, NY, USA, November 13-14, 2019.* ACM, 2019, pp. 306–315. DOI: 10.1145/3360322.3360845.

[LR00] Martin Lauer and Martin A. Riedmiller. "An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems". In: *Proceedings of the 17th International Conference on Machine Learning (ICML 2000).* Morgan Kaufmann, 2000, pp. 535–542.

[LS11] Wenfeng Li and Weiming Shen. "Swarm behavior control of mobile multi-robots with wireless sensor networks". In: *J. Netw. Comput. Appl.* 34.4 (2011), pp. 1398–1407. DOI: 10.1016/j.jnca.2011.03.023. URL: https://doi.org/10.1016/j.jnca.2011.03.023.

[LS20] Shadaj Laddad and Koushik Sen. "ScalaPy: seamless Python interoperability for cross-platform Scala programs". In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Scala.* ACM, Nov.

2020. DOI: 10.1145/3426426.3428485. URL: https://doi.org/10.1145/3426426.3428485.

[MBD18] Yuanqiu Mo, Jacob Beal, and Soura Dasgupta. "An Aggregate Computing Approach to Self-Stabilizing Leader Election". In: *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Trento, Italy, September 3-7, 2018*. IEEE, 2018, pp. 112–117. DOI: 10.1109/FAS-W.2018.00034. URL: https://doi.org/10.1109/FAS-W.2018.00034.

[Mih+12] Mihail Mihaylov et al. "Decentralised reinforcement learning for energy-efficient scheduling in wireless sensor networks". In: *Int. J. Commun. Networks Distributed Syst.* 9.3/4 (2012), pp. 207–224. DOI: 10.1504/IJCNDS.2012.048871.

[Mni+15] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. DOI: 10.1038/nature14236. URL: https://doi.org/10.1038/nature14236.

[Mor+17] Philipp Moritz et al. *Ray: A Distributed Framework for Emerging AI Applications*. 2017. DOI: 10.48550/ARXIV.1712.05889. URL: https://arxiv.org/abs/1712.05889.

[MZL04] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. "Co-Fields: A Physically Inspired Approach to Motion Coordination". In: *IEEE Pervasive Computing* 3.2 (2004), pp. 52–61. DOI: 10.1109/MPRV.2004.1316820.

[NSB03] Radhika Nagpal, Howard E. Shrobe, and Jonathan Bachrach. "Organizing a Global Coordinate System from Local Information on an Ad Hoc Sensor Network". In: *Information Processing in Sensor Networks, 2nd International Workshop, IPSN 2003, , Proceedings*. Vol. 2634. Lecture Notes in Computer Science. Springer, 2003, pp. 333–348. DOI: 10.1007/3-540-36978-3_22.

[Par97] H. Van Dyke Parunak. ""Go to the ant": Engineering principles from natural multi-agent systems". In: *Ann. Oper. Res.* 75 (1997), pp. 69–101. DOI: 10.1023/A\%3A1018980001403.

[Pas+19] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. https://pytorch.org. 2019.

[Pel+20] Giovanni Pellegrini et al. "Learning aggregation functions". In: *arXiv preprint arXiv:2012.08482* (2020).

[Pia+21] Danilo Pianini et al. "Time-Fluid Field-Based Coordination through Programmable Distributed Schedulers". In: *Logical Methods in Computer Science* Volume 17, Issue 4 (Nov. 2021). DOI: 10.46298/lmcs-17(4:13)2021.

[RCR09] Paul Richmond, Simon Coakley, and Daniela M. Romano. "A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA". In: *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*. AAMAS '09. Budapest, Hungary: International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 1125–1126. ISBN: 9780981738178.

[Rey87] Craig W. Reynolds. "Flocks, herds and schools: A distributed behavioral model". In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987, Anaheim, California, USA, July 27-31, 1987*. Ed. by Maureen C. Stone. ACM, 1987, pp. 25–34. DOI: 10.1145/37401.37406. URL: https://doi.org/10.1145/37401.37406.

[Roi+13] Diederik M. Roijers et al. "A Survey of Multi-Objective Sequential Decision-Making". In: *J. Artif. Intell. Res.* 48 (2013), pp. 67–113. DOI: 10.1613/jair.3987.

[Sam+19] Mikayel Samvelyan et al. "The StarCraft Multi-Agent Challenge". In: *CoRR* abs/1902.04043 (2019).

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[SM03] Debashis Saha and Amitava Mukherjee. "Pervasive Computing: A Paradigm for the 21st Century". In: *Computer* 36.3 (2003), pp. 25–31. DOI: 10.1109/MC.2003.1185214. URL: https://doi.org/10.1109/MC.2003.1185214.

[SM16] Subhadeep Sarkar and Sudip Misra. "Theoretical modelling of fog computing: a green computing paradigm to support IoT applications". In: *IET Networks* 5.2 (2016), pp. 23–29. DOI: 10.1049/iet-net.2015.0034. URL: https://doi.org/10.1049/iet-net.2015.0034.

[Sol08] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[Šoš+16] Adrian Šošić et al. *Inverse Reinforcement Learning in Swarm Systems.* 2016. DOI: 10.48550/ARXIV.1602.05450. URL: https://arxiv.org/abs/1602.05450.

[Sos+17] Adrian Sosic et al. "Inverse Reinforcement Learning in Swarm Systems". In: *Proc. of AAMAS.* ACM, 2017, pp. 1413–1421.

[Su+19] Yuhan Su et al. "Cooperative communications with relay selection based on deep reinforcement learning in wireless sensor networks". In: *IEEE Sensors Journal* 19.20 (2019), pp. 9561–9569.

[Sua+19] Joseph Suarez et al. *Neural MMO: A Massively Multiagent Game Environment for Training and Evaluating Intelligent Agents.* 2019. DOI: 10.48550/ARXIV.1903.00784. URL: https://arxiv.org/abs/1903.00784.

[Szu01] Tadeusz Szuba. "A formal definition of the phenomenon of collective intelligence and its IQ measure". In: *Future Gener. Comput. Syst.* 17.4 (2001), pp. 489–500. DOI: 10.1016/S0167-739X(99)00136-3. URL: https://doi.org/10.1016/S0167-739X(99)00136-3.

[Ter+21] J Terry et al. "PettingZoo: Gym for Multi-Agent Reinforcement Learning". In: *Advances in Neural Information Processing Systems.* Ed. by M. Ranzato et al. Vol. 34. Curran Associates, Inc., 2021, pp. 15032–15043. URL: https://proceedings.neurips.cc/paper/2021/file/7ed2d3454c5eea71148b11d0c25104ff-Paper.pdf.

[Tol+19] Ekaterina I. Tolstaya et al. "Learning Decentralized Controllers for Robot Swarms with Graph Neural Networks". In: *3rd Annual Conference on Robot Learning, CoRL 2019, Osaka, Japan, October 30 - November 1, 2019, Proceedings.* Ed. by Leslie Pack Kaelbling, Danica Kragic, and Komei Sugiura. Vol. 100. Proceedings of Machine Learning Research. PMLR, 2019, pp. 671–682. URL: http://proceedings.mlr.press/v100/tolstaya20a.html.

[Tol+20] Ekaterina Tolstaya et al. "Learning decentralized controllers for robot swarms with graph neural networks". In: *Proc. of the Conf. on Robot Learning.* PMLR. 2020, pp. 671–682.

[Tum+04] Luca Tummolini et al. ""Exhibitionists" and "Voyeurs" Do It Better: A Shared Environment for Flexible Coordination with Tacit Messages". In: *Proc. of Int. Workshop on Environments for Multi-Agent Systems.* Springer, 2004, pp. 215–231. DOI: 10.1007/978-3-540-32259-7_11.

[TW12] Karl Tuyls and Gerhard Weiss. "Multiagent Learning: Basics, Challenges, and Prospects". In: *AI Mag.* 33.3 (2012), pp. 41–52. DOI: 10.1609/aimag.v33i3.2426. URL: https://doi.org/10.1609/aimag.v33i3.2426.

[Vir+18] Mirko Viroli et al. "Engineering Resilient Collective Adaptive Systems by Self-Stabilisation". In: *ACM Trans. Model. Comput. Simul.* 28.2 (2018), 16:1–16:28. DOI: 10.1145/3177774. URL: https://doi.org/10.1145/3177774.

[Vir+19] Mirko Viroli et al. "From distributed coordination to field calculus and aggregate computing". In: *J. Log. Algebraic Methods Program.* 109 (2019). DOI: 10.1016/j.jlamp.2019.100486. URL: https://doi.org/10.1016/j.jlamp.2019.100486.

[Wan+19] Minjie Wang et al. "Deep graph library: A graph-centric, highly-performant package for graph neural networks". In: *arXiv preprint arXiv:1909.01315* (2019).

[WT02] David H. Wolpert and Kagan Tumer. "Collective Intelligence, Data Routing and Braess' Paradox". In: *J. Artif. Intell. Res.* 16 (2002), pp. 359–387. DOI: 10.1613/jair.995. URL: https://doi.org/10.1613/jair.995.

[Yan21] Yaodong Yang. "Many-agent reinforcement learning". PhD thesis. UCL (University College London), 2021.

[Yu+21] Liang Yu et al. "A Review of Deep Reinforcement Learning for Smart Building Energy Management". In: *IEEE Internet Things J.* 8.15 (2021), pp. 12046–12063. DOI: 10.1109/JIOT.2021.3078462.

[Zho+20] Jie Zhou et al. "Graph neural networks: A review of methods and applications". In: *AI Open* 1 (2020), pp. 57–81. DOI: 10.1016/j.aiopen.2021.01.001.

[ZYB21] Kaiqing Zhang, Zhuoran Yang, and Tamer Basar. "Decentralized multi-agent reinforcement learning with networked agents: recent advances". In: *Frontiers Inf. Technol. Electron. Eng.* 22.6 (2021), pp. 802–814. DOI: 10.1631/FITEE.1900661. URL: https://doi.org/10.1631/FITEE.1900661.

[ZZL19] Sai Qian Zhang, Qi Zhang, and Jieyu Lin. "Efficient Communication in Multi-Agent Reinforcement Learning via Variance Based Control". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada.* 2019, pp. 3230–3239.

Chapter 15

Conclusion

This dissertation situates itself within the realm of complex systems engineering and many-agent reinforcement learning (ManyRL) applications, adopting unconventional methods to address the unique challenges posed by Cyber-Physical Swarm (CPSW) (introduced in Chapter 2). It is grounded in a language-based approach focused on aggregate computing (discussed in Chapter 3), further integrated with machine learning techniques (overviewed in Chapter 4)—particularly the one based on the notion of *many-agent*.

Focusing on the research questions posed in Chapter 1, we have sought to address the following. Regarding **RQ1**, we have chosen to use the aggregate computing model as a reference because it allows for handling the various challenges of CPSW, including scale, collective behaviors, and the possibility of creating self-healing behaviors. We have also shown other plausible models used in different contexts, such as the multi-agent RL, particularly SwarMDP. We have seen how these can be combined (see Chapter 12).

About **RQ2**, we explored the various aspects of this combination (i.e., the hybrid approach) in Part III. We demonstrated how this seems to be a winning combination at all levels of engineering, both at the application level and the platform level.

For **RQ3**, we sought to outline the main characteristics that differentiate CPSW systems from known ones — see discussion in Chapter 2. In particular homogeneity and self-organisation were the characteristics that then guided the choice of the reference model and the solutions discussed.

Finally, concerning **RQ4**, we discussed throughout the thesis how this influences the design process, discussing new patterns and how this affects the distributed execution platform of such applications.

Following this, an in-depth analysis of the principal contributions and potential future work that continues in the direction proposed by this thesis is presented.

15.1 Discussion

Design Patterns for Cyber-Physical Swarms

AC proves to be a composable, modular, and predictable approach for self-organizing systems, making it versatile across various domain-specific applications. After identifying the class of CPSW, we noted existing gaps in aggregate computing's ability to manage the complexity inherent in these systems. To address this, we first identified algorithms capable of supporting high-level applications, such as the swarm clustering algorithm introduced in Chapter 5. We further sought to identify design patterns that can aid designers in crafting collective processes, as detailed in Chapter 6. Lastly, we encapsulated an essential API for coordinated movement and collective decision-making in Chapter 7. This corpus of contributions serves as a milestone in designing applications for CPSWs, providing designers with both a suite of essential tools and a design space that narrows the gap between the problem space and the solution space.

Deployments in Complex Infrastructure

Aggregate computing offers a flexible model with minimal constraints on deployment. In this dissertation, the model aligns perfectly with the CPSWs systems under consideration. However, this top-down approach eventually needs to be grounded, ensuring that the collective application remains independent of the chosen deployment scheme, thereby decoupling logic from architectural considerations. To that end, we introduced a modern deployment approach for collective applications based on the "pulverization" model and multi-tier programming (Chapter 9). This allowed us to capture essential components required for aggregate computation, facilitating their opportunistic distribution based on the chosen infrastructure, managed through the multi-tier paradigm, particularly leveraging Scala Loci.

Novel Programming Models for CPSws

Field calculus emerged as an effective paradigm for declarative and expressive conceptualization and implementation of complex, self-organizing behaviours at the collective level. This was applied in a range of applications, from swarm robotics and crowd engineering to data centres. However, the approach highlighted certain limitations, particularly concerning *efficiency*– crucial in aggregate systems featuring low-power computational devices. In response, we introduced FRASP (Chapter 8), a new model founded on reactive programming combined with spatial computing, offering greater flexibility in execution by allowing partial re-computation when needed.

Learning for Aggregate Computing

Our language-based approach opened a new frontier in the design of collective applications. Conventionally, a stark distinction exists between manual design (e.g., defining behaviours through programming languages) and automatic design (e.g., employing machine learning techniques for collective behaviours). In this thesis, we attempted to bridge these two worlds, striking a balance between declarative simplicity and the complexity and adaptability of described collective behaviours. To do so, we outlined a roadmap (Chapter 10) that helped us frame the problem and the layers of interaction required for this hybrid vision. Our research ventured into program synthesis in collective systems, where parts of the program were left undefined to be later filled by ManyRL algorithms (Chapter 11). Another line of work focused on learning intelligent scheduling policies to expedite the attainment of equilibrium compared to a pre-set program (Chapter 13). Lastly, we introduce a novel tool called ScaRLib (Chapter 14), which allows for the seamless integration of aggregate computing and deep reinforcement learning, supporting the hybrid design discussed in this thesis.

Aggregate Computing for Learning

In line with the hybrid approach, aggregate computing itself can serve as a medium for enhancing learning. As explored in Chapter 12, we combined graph neural networks with aggregate computing and deep reinforcement learning. This unification proved to be highly effective, allowing for localized learning processes that are informed by collective knowledge gathered through aggregate computing, akin to programmed shimmery.

15.2 Future works

Unified programming model

In the quest for a seamless interface between aggregate computing, novel reactive models, and machine learning, the development of a unified programming model stands as a crucial next step. This framework could enable practitioners to design complex behaviours across the swarm with reduced cognitive load, focusing on higher-level objectives instead of intricate details.

Comprehensive Swarm API

While the presented patterns in this thesis serve as a good foundation, it is by no means exhaustive. Upcoming work aims to expand the API to accommodate more

swarm behaviours and complex orchestrations, further streamlining the application development process for cyber-physical swarm systems. This will entail not only algorithmic contributions but also real-world testing and validation against robustness and scalability criteria.

Benchmarking

As more algorithms and approaches get introduced in this space, benchmarking them against standard metrics becomes essential for meaningful comparison and adoption. Future efforts will aim to establish such benchmarks, enabling the empirical evaluation of different strategies in terms of performance, efficiency, and adaptability.

Opportunistic Deployment

Building on the deployment methods discussed, future work will also investigate opportunistic deployment strategies that take advantage of the system's inherent flexibility. For example, identifying the conditions under which it might be beneficial to move computation closer to the data source or a more powerful computational node, thereby optimizing for both efficiency and latency.

Aggregate Computing for Learning: Distributed Learning

Following the promising results in leveraging aggregate computing for learning, the next steps will focus on enhancing distributed learning paradigms. This can open avenues for more effective, decentralized learning schemes (e.g., federated learning), reducing the need for a centralized learning authority and thereby increasing the robustness and resilience of the system.

Online Learning

The current research has largely focused on offline learning mechanisms. However, real-world applications often require the ability to adapt and learn in *real time*. Future work aims to extend the proposed models and algorithms for online learning capabilities, allowing swarms to dynamically adapt to changing environments.

Learn to Deploy

Taking inspiration from the emerging "DevOps" culture in software engineering, we plan to explore how learning algorithms can aid in the deployment and management of swarm applications. This line of research would aim to automate many

aspects of deployment, from selecting optimal locations for computational nodes to dynamically allocating resources based on real-time system performance.

Unified Design Process

Last but not least, integrating all these disparate elements– programming models, deployment strategies, and learning algorithms—into a unified design process represents a grand challenge for future work. The goal would be to offer practitioners a one-stop solution for designing, deploying, and managing cyber-physical swarm systems effectively and efficiently.

Privacy and Security

Finally, the integration of learning algorithms into swarm systems raises significant concerns about privacy and security. Even if this was not the focus of this thesis, it is essential to consider these aspects in the design of future systems. Future work will need to address these concerns, ensuring that the learning process does not compromise the privacy of individuals or the security of the system as a whole. For instance, *federated learning* can be a promising approach to address privacy concerns, allowing individual agents to learn from local data without sharing it with the central authority.