ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Aggregate Computing and Many-Agent Reinforcement Learning: Towards a Hybrid Toolchain

Tesi di laurea in
PERVASIVE COMPUTING

Relatore
Prof. Mirko Viroli

Correlatore
Dott. Gianluca Aguzzi

Candidato
Davide Domini

Seconda Sessione di Laurea
Anno Accademico 2022-2023

Abstract

The growing popularity of *highly distributed IoT* has highlighted the need for new methods to develop these systems effectively and at scale. Key distinguishing features of these systems include: (*partial observability*) each entity/agent posses only a partial view of the environment in which it operates; (*full distribution*) there is no central entity that coordinates the entire system, as in traditional client-server architectures (instead, computation takes place directly on the IoT device or on some edge devices distributed throughout the system, near the IoT devices); (*uncertainty*) each entity/agent is influenced by its interactions with the environment and with other agents, introducing a level of stochasticity into the system. Over the years, numerous methods have been suggested to address these challenges, including: *Aggregate Computing* [65], a macro-programming paradigm, and *Multi-Agent Reinforcement Learning* [16, 30], a machine learning paradigm. This thesis proposes the starting point for a *hybrid toolchain* that aims to exploit the potential of both aggregate computing and multi-agent reinforcement learning to develop systems capable of learning from experience and self-organizing in case of changes in the external environment.

To attain this objective, we present *ScaRLib* [23], a framework designed to streamline the creation of these systems in simulated settings and JVM-based platforms. ScaRLib focuses on reducing the complexity of development by providing domain abstractions, integration with state-of-the-art tools for multiple subcomponents, a modular and extensible architecture, and a domain-specific language (DSL) to facilitate the configuration of diverse experiments.

Finally, two experiments are also presented to validate the framework functionalities by testing it in basic contexts specific to this domain. These experiments were beneficial in verifying the proper functioning of the tool and highlighting its strengths, as well as identifying areas for future work.

To everyone who has supported me on this long journey.

Acknowledgements

First of all, I would like to thank Professor Mirko Viroli and Gianluca Aguzzi, who have guided and supported me during these months of my first foray into the world of scientific research. Without them, this thesis would not have been possible.

Special thanks also go to my whole family, who have supported me throughout these years, never letting me down and always being there for me.

Finally, I would like to thank all my university colleagues and friends who have stood by my side: Nicolò, Martina, Lorenzo, Ramzi, Filippo, Giacomo, Simone, Alessandro, Andrea, Davide, Enrico, Giulio, Carla, Angela, Chiara, Maria, Chris, Luca and many others.

Contents

Abstract **iii**

1 Introduction **1**

2 Background **3**
2.1 Cyber-Physical Swarms 3
2.2 Aggregate Computing 3
2.3 Reinforcement Learning 10
2.4 Multi-Agent Reinforcement Learning 16
2.5 Simulation 19

3 Requirements **23**
3.1 Domain model 23
3.2 Framework requirements 24
3.3 Scenarios 27

4 Design and implementation **29**
4.1 Framework architecture 29
4.2 Component interactions 33

5 Technologies **37**
5.1 Framework technologies 37
5.2 DevOps technologies used 39
5.3 License 43

6 Validation **45**
6.1 Cohesion and collision 45
6.2 Follow the leader 50
6.3 Discussion 55

7 Conclusions **59**
7.1 Future work 60

List of Figures

2.1 Some examples of Cyber-Physical Swarms. 4
2.2 A random network of devices generated using ScaFi Web. Each purple dot represents a device, and each line represents a neighbour relation. 5
2.3 Field calculus abstract syntax. From [64]. 6
2.4 Aggregate programming abstraction layers. 8
2.5 ScaFi toolkit high level architecture. 9
2.6 A ScaFi AC program example. This algorithm representation is obtained from the code in Listing 2.1 10
2.7 Channel algorithm execution on ScafiWeb. The source is the red dot, the destination is the green dot, the time flows from left to right and from top to bottom. 11
2.8 Agent-environment interactions. 12
2.9 Policy and value based algorithms visual comparison. 13
2.10 Q-Learning and Deep Q-Learning visual comparison 15
2.11 MARL tasks. 17
2.12 MARL execution model comparison. 18
2.13 An Alchemist simulation example. 21

3.1 Some reference scenarios for CPSW. 28

4.1 ScaRLib main modules. 30
4.2 scarlib-core module UML class diagram. 30
4.3 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. 31
4.4 alchemist-scafi module UML class diagram. 33
4.5 UML sequence diagram of the learning process using a CTDE system. 34
4.6 UML sequence diagram of the learning process using a DTDE system. 35

5.1 GitFlow. 40

6.1 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. 46
6.2 Cohesion and collision experiment results. The y-axis represents the reward value. The x-axis represents the total number of timesteps. The first three graphs show the results of the CDTE learning process, while the last three show the results of the DTDE learning process. 50
6.3 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. 51
6.4 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). 51
6.5 Follow the leader training experiment results. The y-axis represents the reward value. The x-axis represents the total number of timesteps. 56
6.6 Snapshots of the learned policy, the time flow is from left to right. . 57
6.7 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. The blue line represents the average distance between the agents, while the orange line represents the average distance between each agent and the leader. 57

Listings

2.1 A ScaFi AC program example. The algorithm implements a channel between a source and a destination. 9

Chapter 1

Introduction

Thesis motivation Significant technological advancements have paved the way for the emergence of a field known as *collective computing* [3], with *Cyber-Physical Swarms (CPSWs)* [53] as a noteworthy branch within it. The latter consist of myriad devices that interact with the environment and exchange information among themselves to reach a *collective* outcome or behaviour. Examples of such systems include swarms of drones, large-scale IoT systems and networks of wearable devices [62, 27]. A crucial aspect of these systems is that a complex collective behavior emerges from the interaction between simpler individual agents leading to adaptive execution of various tasks. Among all aspects related to CPSWs, our focus lies on properties like *collective intelligence* [58] and *self-organization* [52]: hence we shall concentrate on their collective behavior to express autonomy, adaptability, and coordination of the devices that are part of them.

This progress has been driven by research in various related fields such as: multi-agent systems [24], coordination [68], distributed artificial intelligence [14], autonomic computing [41] and many others. Additionally, it has a profound impact on a wide range of applied domains, including: smart cities [70], swarm robotics [15], large-scale IoT systems [61], and many more.

A crucial aspect to consider in CPSWs is how individual devices are programmed and achieve coordination to the perform assigned tasks. Novel approaches – like *aggregate computing* [65] – have focused on manually developing controllers from a global perspective. However, this approach has some drawbacks: it is highly challenging to write satisfactory and efficient programs for complex tasks, they may be error-prone and lack of generality.

On the other hand, approaches exist that leverage various artificial intelligence (AI) techniques, such as *Multi-Agent Reinforcement Learning* (MARL) [16, 30], to enable devices to learn directly from experience and/or data. These approaches have proved to be fundamental to effectively express adaptive behaviors in complex environments and to achieve high performance in a wide range of tasks. However,

they also present several challenges, including: communication, scalability and non-stationarity (i.e., the environment constantly changes) [29].

Thesis objectives The goal of this thesis is to start the design of a *hybrid* approach that can succeed in exploiting the potential of both *macro-programming* and *artificial intelligence* approach. In order to achieve this goal, it is necessary to develop a *toolchain* that allows these systems to be developed in an agile, fast and reusable way. *ScaRLib* [23], is the tool that for us forms the basis of this toolchain. Its main purpose is to integrate *ScaFi* [18] (an implementation of aggregate computing) and *Alchemist* [44] (a general purpose simulator for network oriented systems), with *Reinforcement Learning* used to help the development of experiments in *simulated* environments with *offline learning* (i.e., learning is done once and then the policy is deployed in real-case systems).

Finally, the last goal of this thesis is to validate the functionalities of the framework by implementing two common experiments in this domain; in this way it will be possible to understand the current state of the tool and the work that will be necessary to complete it.

Thesis Structure Chapters 1 and 2 presents a broad scope of the objectives, motivations, and theoretical concepts that serve as the foundation of this thesis. Subsequently, in Chapters 3 to 5, the project that has been developed is presented, starting with the requirements and domain analysis and then delving into the design, implementation, and development process management choices. Finally, in Chapters 6 and 7, two experiments carried out to validate the functionalities of the framework are presented, followed by a discussion of the current state of the project and future work that will be necessary to complete the tool.

Chapter 2

Background

2.1 Cyber-Physical Swarms

Cyber-physical swarms (CPSWs) are an extension of swarm systems in which both logical and physical agents coexist. Swarms draw inspiration from *natural systems* such as ant colonies and bird flocks [57, 50, 13]. In these systems a myriad devices (agents) interact among themselves and with the surrounding environment to achieve a common goal. A fundamental characteristic of these systems is that the local interaction among individual devices usually consists of simple behaviors, but from this a more complex collective behavior emerges that leads to the resolution of the given task. Examples of these systems (Figure 2.1) are a fleet of drones tasked with monitoring a park to oversee adverse events (e.g., fires), or a set of wearable devices to manage crowd congestion in a specific area during a public event.

Swarms have been extensively studied due to a series of significant advantages, namely: i) *cost*: the devices used are typically simpler than a single device that could solve the task individually, resulting in lower costs; ii) *fault-tolerance*: simpler devices are less prone to failures, and there is no single point of failure; iii) *scalability*: the system can be easily scaled by adding or removing devices; iv) *robustness*: the system can handle the loss of some devices; v) *flexibility*: the same system can solve different tasks.

2.2 Aggregate Computing

Context Over the last few years, there has been a definite trend in the development of computational devices: they have been getting smaller, powerful, and less expensive. We have moved from using a single mainframe computer that is shared by an entire department at a company or university, to using numerous

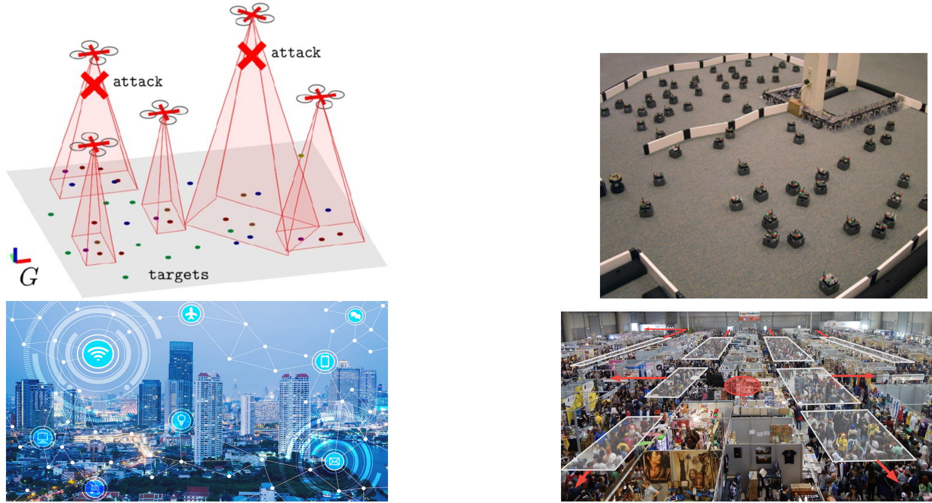


Figure 2.1: Some examples of Cyber-Physical Swarms.

individual devices like smartphones, smartwatches, and others, all interconnected with one another. This phenomenon has led to a range of opportunities, but it has also introduced new challenges when it comes to engineering these *complex distributed systems*. It soon became clear that conventional microprogramming approaches, which involve programming each device separately, lacked of *modularity* and *reusability*. Another problem with microprogramming approaches concerns the difficulty of mapping local behavior of individual agents onto the global behavior of the overall system (*mapping local to gloabl problem*). For this reason, attempts have been made to introduce novel paradigms, including macroprogramming, which shifts focus onto the *collective of devices*, considering their neighbourhood relations and interactions. Some paradigms that have gained significant attention in recent years are *aggregate computing (AC)* [10], *tuple on the air (TOTA)* [39] and *buzz* [45].

Field calculus *Field calculus (FC)* [64] has been designed to be a minimal core calculus aimed at capturing the essential aspects of languages that make use of *computational fields*, somewhat similarly to what λ-calculus does for representing the essence of functional programming. The computational model of FC proposes that a program $\mathbf{P}$ is executed by a network of devices δ, with a concept of *neighborhood relation* representing physical or logical proximity. An example of such a network is shown in Figure 2.2. This notion of neighborhood is used to represent devices capable of direct communication, such as sensors within a broadcast range. In addition to this, the computational model also defines the concept of *computational field* ϕ. These fields [65, 38] are a distributed data structure that

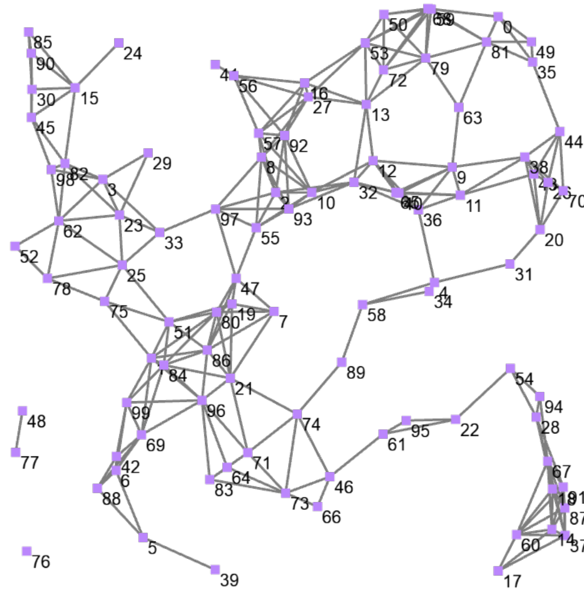


Figure 2.2: A random network of devices generated using ScaFi Web. Each purple dot represents a device, and each line represents a neighbour relation.

maps each device, at a given time, to a value.

A key aspect of FC is that the program (or specification) **P** can be interpreted both *locally* and *globally*. Locally, it is viewed as a description of computation on an individual device, which is executed in asynchronous *computational rounds*. Each round consists of three phases: i) *context building*, each device collects information from the neighborhood and sensors and aggregates it to build its own local context, ii) *program execution*, each device executes the aggregate program on the local context, and iii) *export sharing*, each device shares the result of the computation with the neighborhood. When a device completes a round, it is said "*the device fires*". Globally, on the other hand, a given expression `e` specifies a mapping that associates, for each round of each device, the value `e` assumes at that specific space-time event. This dualism inherently enables the alignment of each device's individual behaviour with the emerging global behaviour of the entire network of devices.

Figure 2.3 gives the basic *abstract syntax* of field calculus. Following this syntax a program P is defined as a sequence of function declarations $\bar{F}$ followed by the main expression e. An expression can be:

- A variable x (e.g., a function parameter)
- A value v. It can be of two types:

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

Figure 2.3: Field calculus abstract syntax. From [64].

- A *local* value l (e.g., a Boolean or an Integer);
- A *neighbouring (field)* value ϕ. It represents a collection of values from neighbors that maps, for each device, the set of neighbour devices δ (including the device itself) to local values l (e.g., a map of neighbours to the distances to those neighbours).

- A function call $f(\bar{e})$ to either a user-declared function or a built-in function;
- A *branching expression* that splits the system into two sub-region depending on how each device evaluates the condition;
- The `nbr` construct, that defines a neighbouring value field ϕ that maps each neighbor with its latest available result of evaluating e;
- The `rep` construct, which models state evolution over time.

In addition to the syntax explained for FC, there is an expanded version for *higher-order FC*. Here, functions are considered *first-class values* and programmers are able to pass functions as parameters to other functions. This allows for the addition or modification of existing code within the network.

One final important aspect of field calculus is its ability to formally demonstrate the validity of significant properties, including:

- *self-stabilization* [63, 66, 22, 36]: this feature serves to prevent the system from entering incorrect states. Firstly, it guarantees that, given a costant input, a program's execution will converge to a specific value in a finite time for each device. Secondly, it ensures that this value is solely dependent on the input and not on previous execution history.
- *global coherence*: this characteristic is assured by the field calculus, which ensures consistent alignment of the *nbr* operator across the network. However,

achieving this isn't straightforward, as multiple requests for *nbr* can occur, and the execution speed of functions can vary depending on the device. If global-local coherence isn't maintained, there may come a point where a subset of network devices loses synchronization with the entire network.

- *space-time universality* [7]: this property guarantees that field calculus is computationally universal (or Turing complete [59, 60]) and that it can be used to solve any computable problem.

- *eventual consistency* [11]: this property is closely related with self-stabilization. It approximates the network of devices as a continuous environment; a system is eventual consistent if the state it converges to is set by that continuous environment rather than the particulars of how devices are distributed through it.

Agregate computing *Aggregate computing* [10] is a macro-programming model whose foundation is based on *field calculus*, providing innovative solutions to the issues previously mentioned. AC's objective is to reduce the complexity of designing, developing, and maintaining complex distributed systems, with focus on three fundamental aspects: i) the composition of modules and subsystems must transparent, ii) different subsystems needs different coordination mechanisms for different regions and times, and iii) the implementation details of coordination mechanisms should be hidden from programmers in order to facilitate ease of use. To attain this objective, AC adopts a *layer-based architecture* (Figure 2.4) that, in addition to field calculus, includes two levels of abstraction to increase resilience and hide complexity.

The first additional layer, placed immediately above the FC layer, is crucial for hiding complexity and supporting the efficient engineering of distributed coordination systems. In practice, it introduces various operators that can be used by the subsequent levels as building blocks, namely: i) **G** allows the spread of information from a source to a given distance, ii) **C** is the inverse of G, allowing the collection of information, iii) **T** allows to keep track of the time, and iv) **S** allows the election of a set of leaders and the partition of the network according to those leaders. The second-to-last layer comprises libraries and features tailored towards offering a simple interface for software developers. Furthermore, this layer encourages reusability, productivity, declarativeness, flexibility, and efficiency. An instance of a robust API is *ScaFi* [18], a Domain-Specific Language for implementing aggregate computing via the Scala programming language. Finally, there is the application layer, where the end-user can integrate their own services based on aggregate computing, such as a crowd monitoring system in a smart city.

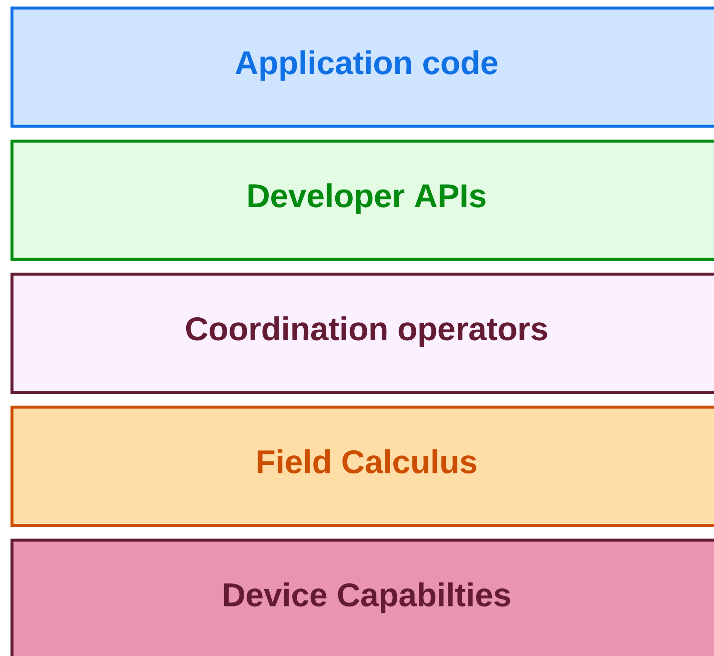


Figure 2.4: Aggregate programming abstraction layers.

ScaFi *ScaFi* [18] is a toolkit for the Scala language that provides a domain-specific language, libraries, a simulation environment and runtime support for developing aggregate computing based systems.

The architecture of ScaFi consists of various modules (Figure 2.5), the main ones being: i) `scafi-core`, which contains the DSL and a standard library of reusable functions (e.g., Gradients, BlockG and BlockS), ii) `scafi-stdlib-ext`, which provides a set of additional functions; as these require external dependencies, it is kept separate from the core, iii) `scafi-simulator`, which offers support for simulating aggregate systems, iv) `scafi-simulator-gui`, which provides a GUI for visualizing and interacting with the simulation, v) `spala`, which provides an actor-based middleware for AC based on Akka [31], vi) `scala-distributed`, which provides an integration layer for spala in ScaFi.

An example of how ScaFi works is shown in Figure 2.6 and Listing 2.1, the goal is create a channel given a source, a destination and a width. To reach this goal is it possible to define the computation as a pure function over fields that exploits and composes the following functions: i) `gradient`, which takes as input

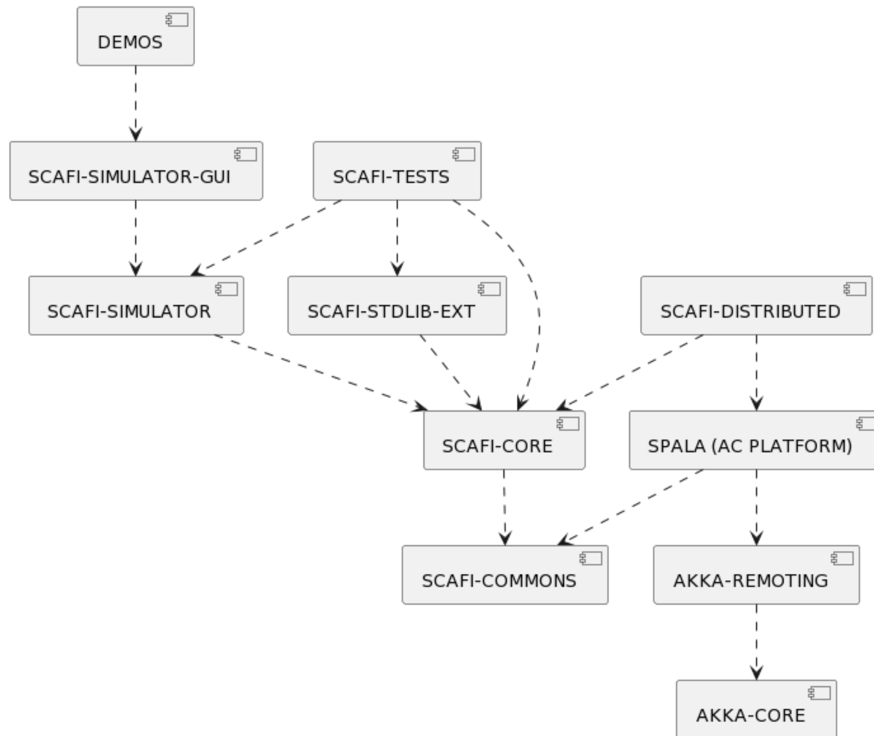


Figure 2.5: ScaFi toolkit high level architecture.

a field of Booleans and computes in output a field of Integers with the mininum distance, for each point, from a given source represented by the values set to true in the input; ii) `distance`, which computes the distance between two sources, and iii) `dilate`, which takes as input a field of Booleans and stretches the source by a given width.

Figure 2.7 shows the execution of the channel algorithm on ScaFiWeb.

Listing 2.1: A ScaFi AC program example. The algorithm implements a channel between a source and a destination.

```
def channel(
    source: Boolean,
    destination: Boolean,
    width: Double
): Double {
    dilate(gradient(source) + gradient(destination)
        <= distance(source, destination), width)
}
```

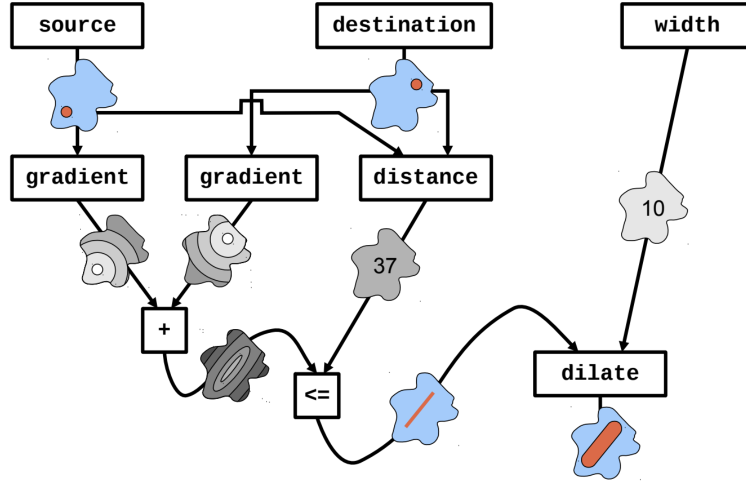


Figure 2.6: A ScaFi AC program example. This algorithm representation is obtained from the code in Listing 2.1

.

2.3 Reinforcement Learning

"*Reinforcement Learning (RL)* is the science of decision making. It is about learning the optimal behavior in a environment to obtain maximum reward"[1]. RL is a general framework, other than supervised and unsupervised learning, in which an *agent* learns to behave within an *environment* by performing some *actions* and seeing the result they produce. It is inspired by how humans and animals learn through the system of rewards and punishments: for each good action the environment provides to the agent a positive reward, instead, for each bad action the agent gets a negative reward (also called penalty).

Formally, a RL problem can be formulated as following [35]:

- Discrete time steps $t = 0, 1, 2, ...$;
- A discrete set of environment states $\mathcal{S}$;
- A discrete set of agent actions $\mathcal{A}$;
- A reward signal;
- A probabilistic policy π, that is a mapping function from states to actions.

The goal of the agent is to learn the optimal policy π^* in order to maximize some long-run measure of reinforcement (e.g., the Infinite Horizon Discounted Model

[1] https://www.synopsys.com/ai/what-is-reinforcement-learning.html

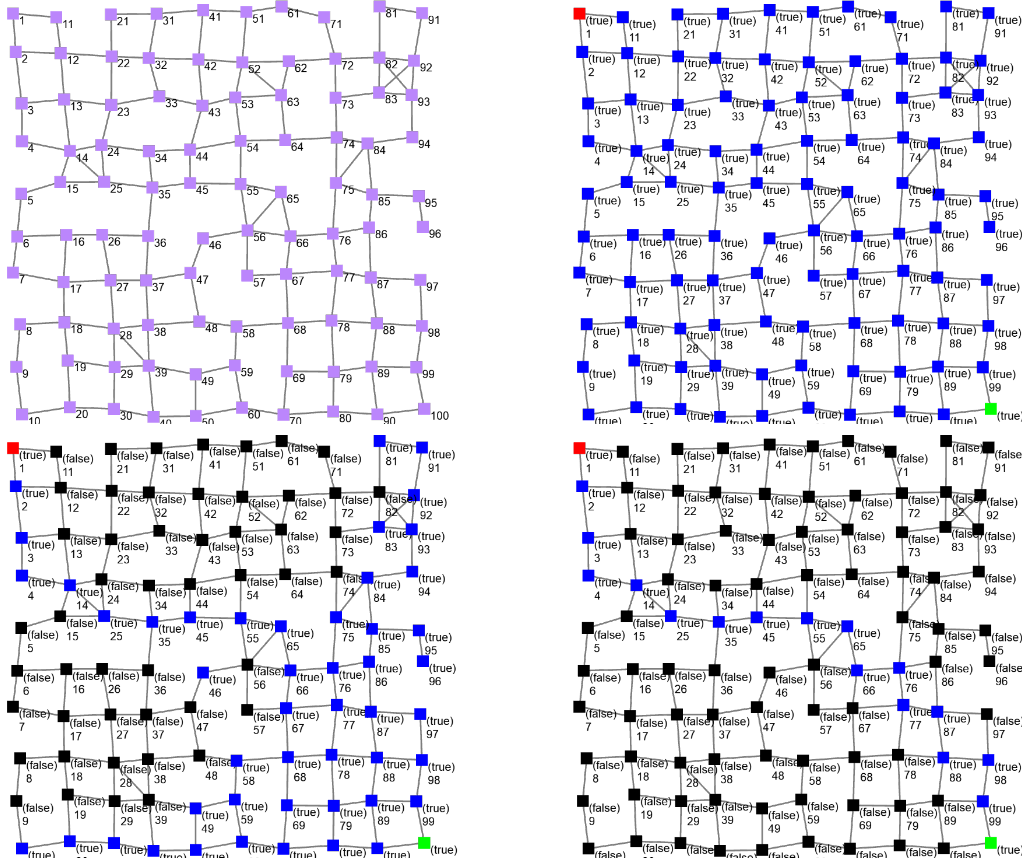


Figure 2.7: Channel algorithm execution on ScafiWeb. The source is the red dot, the destination is the green dot, the time flows from left to right and from top to bottom.

[35]). First, at time t, the agent observes the state of the environment $s_t \in \mathcal{S}$ and chooses an action $a_t \in \mathcal{A}$ using the actual policy π_t. Thereafter, the environment: takes in the action a_t, emits the new state $s_{t+1} \in \mathcal{S}$ and returns the scalar reward r_{t+1} (Figure 2.8). Finally, the agent, based on the reward obtained updates its knwoledge. This formulation stems from *Markov Decision Processes* [28, 46], which is a mathematical framework for *sequential decision making*. A very important property that these systems must adhere to is the *Markov property*, which states "*the future is independent of the past given the present.*" In other words, it implies that the state transition function does not require the entire past trajectory but only the last state, namely:

$$p(s_{t+1}|s_t, a_t, ..., s_0, a_0) = p(s_{t+1}|s_t, a_t)$$

It is important to note that the concept of state can always be extended to satisfy

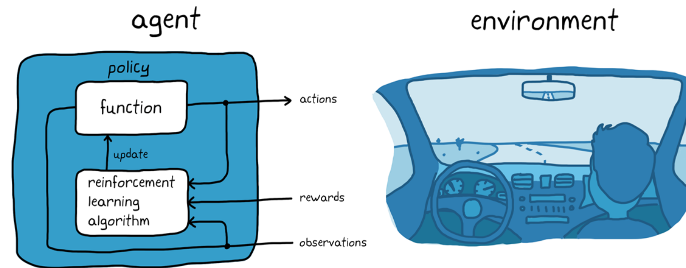


Figure 2.8: Agent-environment interactions.
Source: https://it.mathworks.com/discovery/reinforcement-learning.html

this property.

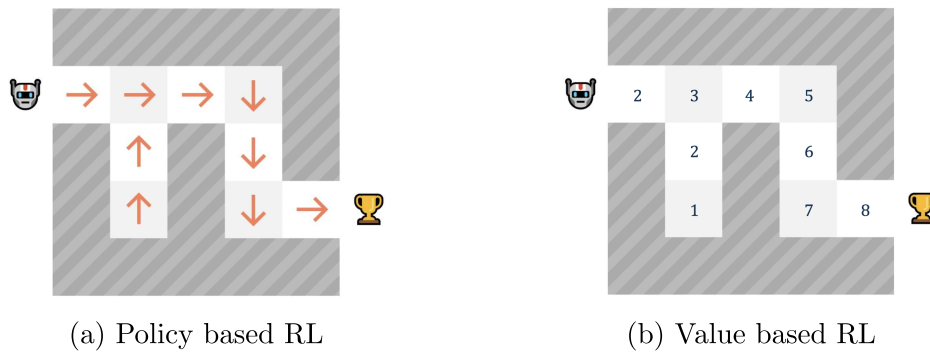
In order to find the optimal policy π^*, the agent tries to maximize the expected cumulative reward. Since the environment is stochastic (i.e., the same action performed in the same state could lead to different results over time) the more you look into the future the more the outcome could diverge. For this reason, it is common to use a model that takes less account of rewards that are far away in time than those that are close in time:

$$R_t = \sum_{i=t}^{\infty} \gamma^{i-t} \cdot R_{\pi(s_i)}(s_i, s_{i+1})$$

This model is called *Infinite Horizon Discounted Model*, the key aspect is the hyper-parameter γ. It is a scalar weight in the range $[0; 1]$, in this way, the further away the reward is in time, the smaller its weight.

Exploration-exploitation dilemma The *exploration-exploitation dilemma* is a problem that comes from the definition of the RL process. In order to increase its knowledge and build an optimal policy, the agent needs to *explore* the environment in the hope of finding better actions. After some exploration, the agent might have found a set of apparently rewarding actions, but, how can the agent be sure that the found actions are actually the best? When should the agent continue to explore or else, when should it just *exploit* its existing knwoledge?

Several exploration strategies have been proposed in the literature to solve this problem, the simplest is the ϵ-*greedy* strategy. The agent *randomly explore* the environment with probability ϵ while *exploit* the current optimal action with probability $1 - \epsilon$.



(a) Policy based RL

(b) Value based RL

Figure 2.9: Policy and value based algorithms visual comparison.
Source: https://www.lamsalashish.com.np/blog/reinforcement-learning

$$\pi(s) = \begin{cases} \pi^*(s) & \text{with probability } 1-\epsilon \\ \textit{random action} & \text{with probability } \epsilon \end{cases}$$

Usually, at the beginning of the learning process ϵ starts near to 1 (i.e., more exploration) and then decreases to 0 as the agent learns more and more about the environment.

Main approaches In Reinforcement Learning, there are two main families of approaches that can be used to categorize the algorithms employed by an agent in finding the optimal policy. These are: i) *policy-based* methods, and ii) *value-based* methods. Policy-based algorithms aim to directly learn a function that maps each state to the best action to take (or a probability distribution over a set of possible actions). On the other hand, value-based methods seek to learn a function that maps each possible state to an expected value of being in that state. This way, the agent can learn which states is more valuable and will take action that leads to it. This comparison is well illustrated in Figure 2.9.

An example of policy-based method is *Proximal Policy Optimization (PPO)* [54], while an example of value based-method is *Q-Learning* [67]

Q-Learning *Q-Learning* is one of the most famous Reinforcement Learning algorithm from the value-based methods family. One of the key aspects of this algorithm is the *Q-Table*, denoted as $Q(s, a)$. This table represents, for each possible state-action pair, the *expected cumulative reward* that the agent will obtain by taking action a in state s and subsequently following optimal actions. Thus, the Q-table at time step t, given a state s_t, an action a_t, and a policy π_t, is represented by:

$$Q(s_t, a_t) = max_{\pi(s_t)=a_t} R_{t+1}$$

Starting from the Q-Table, it is possible to define the *optimal policy* as follows:

$$\pi^*(s) = argmax_a Q^*(s, a)$$

Another key aspect is how we can estimate the reward at the end of the process if we only know the current state and action, without knowing the subsequent trajectory. To achieve this, the *Bellman equation* can be used. This equation defines the value $Q(s, a)$ recursively as the sum of the immediate reward and the maximum expected cumulative reward from the subsequent state:

$$Q(s_t, a_t) = r_{t+1} + \gamma \cdot max_a Q(s_{t+1}, a)$$

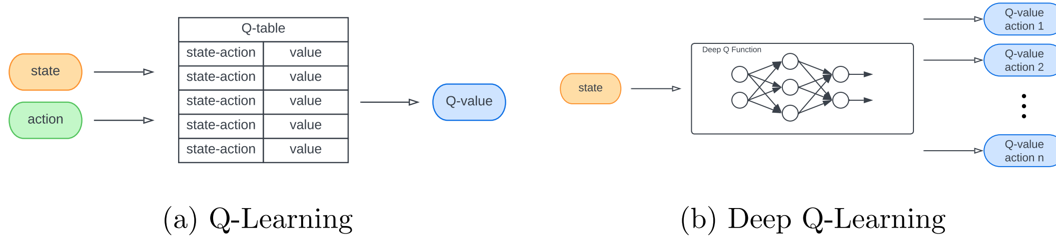
The main idea of Q-Learning is to *iteratively approximate* the Q-values, using the Bellman equation, as follows:

$$\begin{aligned} Q(s_t, a_t) = & \underbrace{(1-\alpha)}_{\text{Learning Rate}} \cdot \underbrace{Q(s_t, a_t)}_{\text{Old Value}} \\ & + \underbrace{\alpha}_{\text{Learning Rate}} \cdot \overbrace{\Big(\underbrace{r_{t+1}}_{\text{Reward}} + \underbrace{\gamma}_{\text{Discount Factor}} \cdot \underbrace{max_a Q(s_{t+1}, a)}_{\text{Maximum Future Reward}} \Big)}^{\text{Learned Value}} \end{aligned} \tag{2.1}$$

Where α is the learning rate hyper-parameter that controls how much of the current Q-value and newly proposed Q-value is considered. At the beginning of the learning process, these Q-values will be practically random estimates and may be completely wrong. However, it has been demonstrated that as iterations progress, these Q-values will converge and represent the true Q-values.

Deep Reinforcement Learning Classical algorithms of reinforcement learning, when applied in *real-world contexts*, suffer from the problem of *state space explosion*. This arises from an exceedingly large number of possible states, making the resolution of a given task *computationally intractable*. For instance, in the game of chess, there can be around 100^{120} possible typical games, a number much larger than the count of sand grains on Earth ($\approx 10^{23}$) and the number of atoms in the observable universe ($\approx 10^{81}$). For this reason, *deep reinforcement learning* has been introduced, which involves utilizing deep neural networks as approximators for the policy and/or the value function.

One of the most well-known Deep RL algorithms is DQN [40]. This was developed by DeepMind in 2013 and was initially used to train an agent capable of playing Atari video games. One of the advantages of using neural networks as approximators for the function to be learned is the ability to avoid hand-engineering the state space and instead allow the network to learn the best features directly.



(a) Q-Learning (b) Deep Q-Learning

Figure 2.10: Q-Learning and Deep Q-Learning visual comparison

For example, in the case of Atari games, this is achieved by using convolutional layers and feeding the network with screen raw pixels.

DQN, specifically, is the deep version of the Q-Learning algorithm, thus producing a Q-Value output for each possible action (Figure 2.10). This makes the training of the utilized neural network a regression task in which a squared error loss can be employed as the loss function:

$$L = (y_t - \hat{y}_t)^2$$

Where y_t represents the actual value and $\hat{y}_t$ is the predicted value at time t. Since RL is an unsupervised learning, and therefore labels are not available, the value y_t is estimated using the Bellman equation, transforming the loss function for a transition $< s_t, a_t, s_{t+1}, r_{t+1} >$ into:

$$L = (r_{t+1} + \gamma \cdot max_a Q(s_{t+1}, a) - Q(s_t, a_t))^2$$

When attempting to use neural networks to approximate the Q-function, several problems arise. The first one is due to the high correlation that exists between two consecutive transitions within the same episode. This correlation leads to a significant decrease in variance, causing the network to tend to forget previous transitions as it overwrites them with newer ones. For instance, let's consider an agent's task to learn to play a level-based video game; this issue implies that while the agent tries to learn how to navigate the second level, it might forget how to behave in the first level. The most common solution is to employ an *experience replay* (i.e., a buffer), where all transitions $< s_t, a_t, s_{t+1}, r_{t+1} >$ are stored. When updating weights, a random mini-batch is sampled from this buffer, breaking the correlation between consecutive transitions. Furthermore, since a transition can be used in multiple weight updates, this approach also improves data efficiency.

A second issue that can be observed is referred to as the *moving target problem*. This stems from the fact that, when updating the network's weights, both the predicted values and the target values are estimated using the same neural network. This leads to a strong correlation between the target values and the network's weights, introducing significant oscillations during training. To address this

problem, two distinct neural networks with identical architecture are employed: i) *the action network* Q, used to determine the agent's actions, which is updated every u steps; ii) *the target network* $\hat{Q}$, used to calculate the target values, updated every c steps. Typically, $c >> u$, and the *target network's* weight update involves replacing the existing weights with those from the *action network*. Since the target values are generated using an older set of weights, a delay is introduced between the moment the Q network is updated and the moment it starts to affects the target values. This delay reduces the likelihood of divergence and oscillations. The loss function becomes:

$$L = (r_{t+1} + \gamma \cdot max_a \hat{Q}(s_{t+1}, a) - Q(s_t, a_t))^2$$

2.4 Multi-Agent Reinforcement Learning

Multi-Agent Reinforcement Learning is an extension of RL where multiple *learning agents* interact one another and with the environment. Usually, MARL is modelled as a *Markov Game* (or Stochastic Game $\mathcal{S}$) [37] in which we have:

- A tuple $\mathcal{S} =< N, S, \{A^i\}, P, \{R^i\} >$ with $i \in 1 \dots N$
- The number of agents $N > 1$
- The action space of the i-th agent A^i. The global action space is defined as $\mathbb{A} = A^1 \times A^2 \times \dots \times A^N$
- A function describing the transition dynamics $P : S \times \mathbb{A} \rightarrow \mathcal{P}(S)$
- The reward function $R^i : S \times \mathbb{A} \times S \rightarrow \mathbb{R}$ for each agent i

Categorization Based on the reward function used by the agents, MARL can be divided into three categories (Figure 2.11): i) *cooperative*, where all the agents trying to maximize the same reward function (e.g., a group of robots trying to clean a room); ii) *competitive*, where, potentially, each agent has its own reward function that is conflictual with the other (e.g., a rock-paper-scissor game). iii) *mixed*, where some agents are cooperative and others are competitive (e.g., a soccer game). Cooperative MARL, with respect to the policy, can be further divided into two additional categories, namely: i) *homogeneous*, where all the agents have the same capabilities, i.e., they use the same policy ii) *heterogeneous*, where each agent may have its own policy, in this case, each agent tries to maximize the local policy following the global shared goal.

In this thesis, we focus on a subset of MARL, namely: *Many Agent Reinforcement Learning* [69]. The only difference between the two approaches is in the

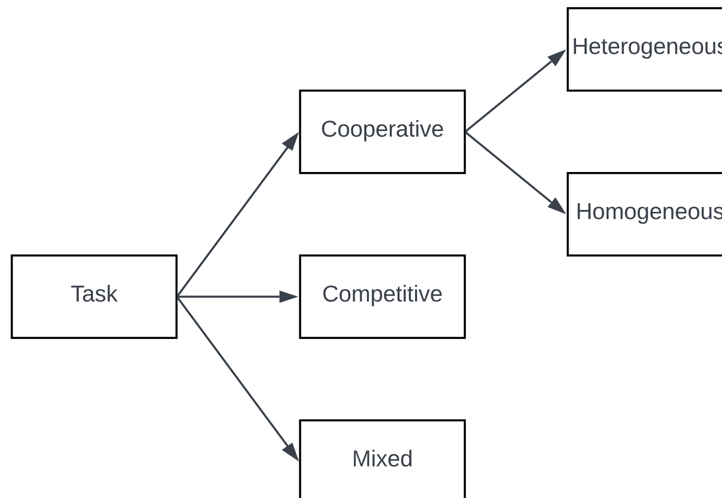
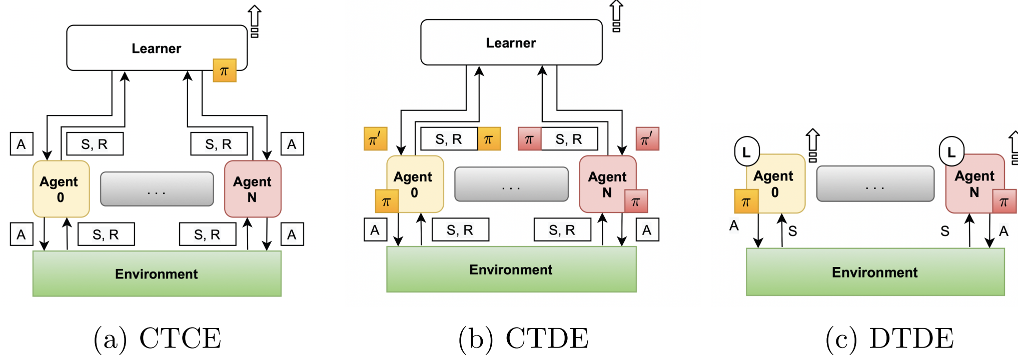


Figure 2.11: MARL tasks.

number of agents involved. Typically, in Many Agent Reinforcement Learning the number of agents may range from a hundred to one or two thousands whereas, in Multi Agent Reinforcement Learning, there are only a few tens [51, 9]. Moreover, we focus on cooperative homogeneous and heterogeneous learning.

Training and execution model Another point to pay attention to is the system by which the training and execution of the various agents are carried out. The main question is: *what information is available to agents during training and after the learning of policies?* Thus, MARL algorithms can be categorized into three types (Figure 2.12):

1. *Centralized Training Centralized Execution (CTCE)*, in this type of system, there is a higher-level agent called *Learner*, with a global perspective, whose task is to perform training and compute actions to be undertaken. The remaining agents, therefore, transmit their local environmental perceptions (i.e., the local state) to the learner agent, which is responsible for merging these perceptions to reconstruct the global state. Once reconstructed, it selects the action to take in accordance with the current policy and evaluates the reward function for policy updates. This system reduces the multi-agent game to a single-agent game using the history of observation of all agents, which can be useful in context with partial observability or where complex coordination between agents is required. However, there are three main



(a) CTCE (b) CTDE (c) DTDE

Figure 2.12: MARL execution model comparison.

drawbacks: i) the central policy uses the joint action space, which usually grows exponentially with the number of agents, ii) the environment assigns a collective reward to all agents, and it may prove challenging or unfeasible to convert it into an individual reward for each agent, and iii) as the agents are distributed, they may be unable to communicate with a central learner. For instance, transmitting all sensor and camera data in autonomous cars is unfeasible.

2. *Decentralized Training Decentralized Execution (DTDE)*, in this type of system, each agent has its own local policy and can only observe a portion of the environment. Each agent, therefore, takes actions based on its perception of the environment and updates its own local reward function accordingly. The primary advantage of this approach is its ability to prevent exponential growth of the action space, although it does have some disadvantages, including: i) as agents are trained concurrently, they may be significant affected by non-stationarity, and ii) agents' policy cannot rely on shared information neither during training nor during execution.

3. *Centralized Training Decentralized Execution (CTDE)*, in this type of system, a learner agent with a global perspective executes the training algorithm. Once the policy is updated, it is sent to each agent, which can interact with the environment and take actions in accordance with it. This system combines the benefits of the previous two approaches.

Challenges MARL is a highly promising field that has been garnering increasing attention in recent years. Nevertheless, there are still several challenges that complicate its application in complex contexts. Some of these challenges include:

1. *Partially Observable Environments*: in environments of significant complexity, agents cannot have a complete representation of the surrounding environment nor access the state of other present agents.

2. *Non-Stationarity*: this issue arises due to multiple agents simultaneously learning and changing the environment. From the perspective of an individual agent, the environment becomes non-stationary.

3. *Agent Communication*: communication among agents is a crucial feature in the literature. The problem not only addresses what an agent should communicate but also with whom and when.

4. *Coordination*: coordination is essential in cooperative systems, as agents must reach a consensus on the actions to take. Failure to coordinate during learning can lead to suboptimal policies. Coordination can be achieved through communication or implicitly, with each agent constructing its own model of other agents' behavior to infer their next actions.

5. *Multi-Agent Credit Assignment Problem*: this refers to associating a reward with an action taken by a specific agent. This association is crucial for learning to understand the effectiveness of an action and maximizing the reward function over time.

6. *Scalability*: the scalability of these systems is influenced by the challenges outlined in this section. Training a single agent is inherently challenging; as the number of agents in the system increases, the complexity grows exponentially. Additionally, scalability is determined by an agent's robustness in the face of changes in the behavior of other agents. Literature presents exploration techniques to enhance scalability, such as knowledge reuse [21], regularization [49, 56], and others.

2.5 Simulation

In science and engineering, the use of *simulators* plays a key role. These simulators enable the experimentation with complex and expensive systems (such as cyber-physical swarms), offering a virtual environment in which to conduct tests, analyses, and in-depth studies rapidly, under controlled and repeatable conditions (e.g., [6, 8, 20, 17]). Moreover, simulation is crucial in MARL with offline learning, enabling agents to learn a policy for a model of the environment without relying on physical components, such as robots, that may be cost-prohibitive or unavailable. In this project, the *Alchemist* [42] simulator has been taken as a reference.

Alchemist is a *meta-simulator* designed for simulating *complex distributed systems* in a rich variety of scenarios like swarm robotics [19], large-scale sensor networks [4], crowd simulation [10], path planning, and even morphogenesis of multi-cellular systems.

This simulator is "meta" *by design*, this stems from the fact that it is based on general abstractions that can be mapped to specific use cases (i.e., *incarnations*). The meta-model is inspired by biochemistry and 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 a variety of use cases and node numbers (it could support thousands of nodes), while maintaining a consistent underlying structure

Alchemist features four incarnations: biochemistry, sapere, protelis, and ScaFi, each with a different way of modeling molecules and actions. In this project, the *ScaFi* incarnation has been taken as a reference. It supports the ScaFi Scala DSL and has been used in distributed peer-to-peer chats and situated problem-solving

Alchemist offers an straightforward method for defining simulations. The process requires a YAML file that includes essential parameters, such as the incarnation type, neighbor connection model, and node deployment. In Figure 2, we have provided an example YAML file that creates a simulation using the ScaFi incarnation (first row). It also defines the neighborhood 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.

```
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

```

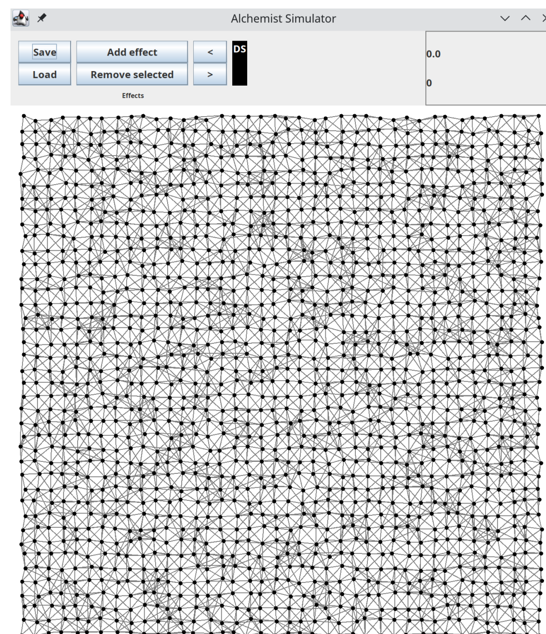


Figure 2.13: An Alchemist simulation example.

Chapter 3

Requirements

This chapter first introduces the domain model of the *hybrid AC-MARL* approach to engineer CPSWs, and then presents the framework requirements. Finally, some scenarios are explained to provide a reference context.

3.1 Domain model

The proposed *hybrid* approach between *AC* and *MARL* aims to combine the strengths of the two approaches to engineer CPSWs. The idea is to leverage AC to define *structural* rules of the swarm (e.g., leader election [43]) while using MARL to define *behavioral* rules (e.g., the choice of an action by an agent). Harnessing machine learning allows for the definition of more complex and adaptive behaviors, which would be challenging to achieve through programmatic methods.

To accurately define the domain, a *Domain-Driven Design* [26] approach was used. First, the *ubiquitous language*, as presented in Table 3.1, was defined. This associates a precise definition with each term of the domain, thereby eliminating potential *ambiguities*, which are common given the breadth of the MARL domain.

The *environment* is the context in which *agents* are immersed and operate, cooperating to solve the assigned task. Agents interact with the environment by receiving the *state* it is in before choosing which *action* to perform, but they are also equipped with *sensors* through which they can perceive certain features of the context they are in, such as the number of other agents in the neighborhood and the distance from them. Each agent chooses which action to take based on its own behavior *policy*. This policy is updated in accordance with the *system* that encompasses the agents themselves and the environment. This system is one of the key abstractions and defines the execution flow of agents. For example, in a *CTDE (Centralized Training and Decentralized Execution)* system, agents rely on a centralized learner that updates the policy as experience is accumulated

and then also distributes the new policy to each agent. On the other hand, in a *DTDE (Decentralized Training and Decentralized Execution)* system, each agent is responsible for its own policy, potentially resulting in each agent having a different policy from all the others.

3.2 Framework requirements

This section describes the requirements that the project must satisfy.

Business requirements

The business requirements specify the characteristics that the system must have to be correct. Those identified for ScaRLib are:

- The framework must enable the development of CMARL systems for JVM-based environments through a high-level specification;
- The framework should support different training and execution models;
- The framework should be extensible and modular to allow the integration of:
  - different learning algorithms;
  - different simulators;
  - different deep learning frameworks.

User requirements

The user requirements express the needs of the users and describe the actions that the user must be able to perform by interacting with the system. From the previous domain analysis that was carried out, we can identify the following user requirements:

- It should be possible to configure the learning system through a *DSL*;
- It should be possible to define a custom *environment*;
- It should be possible to define a custom *state space*;
- It should be possible to define a custom *action space*;
- It should be possible to define a custom *reward function*;

Concept	***Definition***
Environment	The context in which the *agents* are immersed and operate, it is a representation of the task to be solved. It is capable of interacting with the agents, providing information about its current *state*, receiving the *actions* that one or more agents wish to execute, and returning the corresponding *reward*.
Agent	An entity that interacts within an *environment* and with other *agents* in order to learn the optimal *actions* sequence to maximize a *reward* signal. It is equipped with *sensors*, *actuators* and a *communication mechanism*.
State	A representation of the *environment* at a given time, including any relevant information that an *agent* can perceive or use to make decisions about its *actions*.
Action	A decision or choice made by an *agent* in response to the current state of the *environment*.
System	A collection of *agents* that interact within a shared *environment*. It defines the training and execution flow of the agents.
Policy	A function that maps the current *state* of the *environment* to a probability distribution over the set of possible *actions* that the *agent* can take in that *state*. The policy specifies the *agent's* behavior or strategy in response to different *states* of the *environment*, and it is learned through a process of trial and error using the *reward* signal as feedback.
Sensor	A device that allows an *agent* to perceive certain characteristic of the *environment*.
Actuator	A mechanism that allows an *agent* to perform certain actions on the *environment*.

Table 3.1: Hybrid AC-MARL approach ubiquitous language

- It should be possible to define a custom *neural network* to approximate the policy/value-function;
- It should be possible to define some of the agent logic through *aggregate programming*;
- It should be possible to log information related to the training process;
- It should be possible to visualize the execution of agents, both during training and testing.

Functional requirements

The functional requirements relate to the functionalities that the system must make available to the user. To define them, it is necessary to rely on the user requirements extracted previously.

- The framework must allow the user to define their own *experiment*, this includes: i) the environment, ii) the state space, iii) the action space, and iv) the reward function;
- The framework must allow the user to define their own *learning algorithm*;
- The framework must allow the user to define their own *neural network* to approximate the policy/value-function;
- The framework must allow the user to define their own *agent logic* through aggregate programming;
- The framework must allow the user to log information related to the training process;
- The framework must allow the user to visualize the execution of agents, both during training and testing.

Non-functional requirements

Non-functional requirements concern the functionalities that the system does not necessarily have to possess in order to ensure that it is correct. The following non-functional requirements have been identified within the system to be developed:

- The framework should provide an easy and clean API;
- The framework must be cross-platform, therefore it must be executable on any operating system capable of supporting Java version 11 or later;

- The framework should be extensible and modular, allowing the user to customize some of its components, along the following dimensions: i) learning algorithms, ii) simulators, and iii) deep learning frameworks.

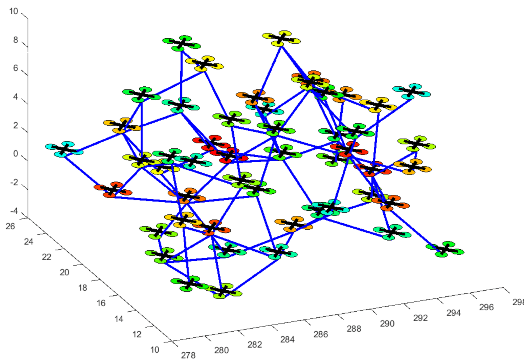
3.3 Scenarios

This section aims to provide a *reference context* for the use of the framework by attempting to illustrate, through some practical examples, the key characteristics within the domain of CPSWs.

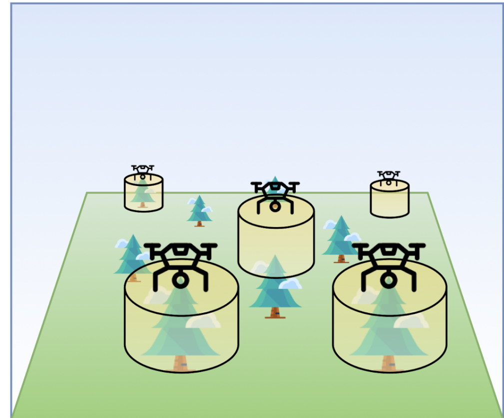
The first scenario (Figure 3.1a) represents one of the simplest conceivable instances for CPSWs, in which the proposed hybrid approach can be beneficial. It involves a *fleet of drones*, each drone having a predetermined *set of neighbors*. The objective for each drone is to maintain proximity to its neighbors (i.e., *cohesion*), while avoiding *collisions*. Each drone can only observe a limited portion of the environment (i.e., *partial observability*), which comprises only its neighboring drones and not the positions of the entire fleet. The goal is to define a policy that allows the fleet to form many clusters, in which each drone has a mean distance from its neighbors that is as close as possible to a given threshold. This first scenario will be implemented in Chapter 6 to validate the framework.

The second example (Figures 3.1b and 3.1c) also involves a fleet of drones but is more intricate than the previous one. In this case, the fleet's purpose is to monitor a designated area of territory for *adverse events* (e.g., fires). Once an adverse event is identified, the fleet must *coordinate* to determine the number of drones to intervene and their respective *strategies*. The goal is to define a policy that allows the fleet to identify the adverse events and coordinate to handle them within a given time frame.

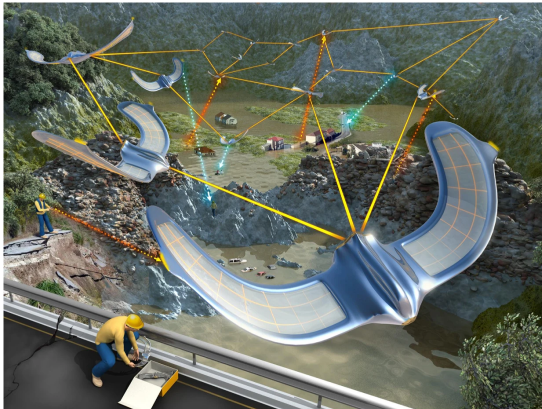
A final example (Figure 3.1d) entails the use of *wearable devices* (e.g., smartwatches) for *crowd management* during a public event (e.g., a conference or concert). In this scenario, the agents constituting the swarm are the individuals themselves, as represented by their wearable devices. The goal is to prevent the formation of overly *crowded* clusters and to determine the optimal routes for evacuating the area in the event of an emergency.



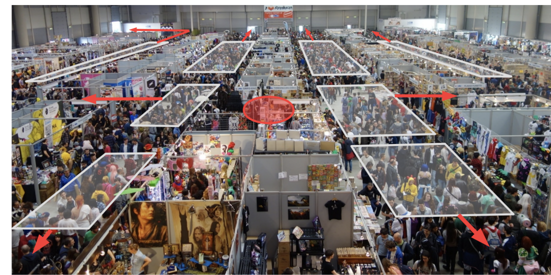
(a) Drones flocking [34].



(b) Adversial events monitoring.



(c) Adversial events monitoring.



(d) Crowd management.

Figure 3.1: Some reference scenarios for CPSW.

Chapter 4

Design and implementation

This chapter presents the design decisions for modelling the domain and implementing the framework. It introduces the key details of each module, followed by a discussion of the interactions between them.

4.1 Framework architecture

The framework has been devised to aid the development of CMARL systems in JVM-based environments through high-level specifications. For this purpose, the tool is divided into three primary modules (Figure 4.1), namely: i) `scarlib-core`: which captures the core concepts of the CMARL domain by abstracting low-level implementation details, ii) `dsl-core`: which provides a high-level language for specifying a CMARL system, and iii) `alchemist-scafi`: which provides bindings between ScaRLib and the Alchemist and ScaFi tools, enabling experiments in a simulated environment using aggregate computing. One aspect to consider is the `pytorch` sub-module upon which `scarlib-core` relies. This sub-module serves as a *learning engine* to conduct training and optimization of neural networks utilised by the RL algorithms. It's worth noting that, if required, an alternative substitute module (e.g., DL4J [1]) that has similar functionality can be used instead.

The modularization of the framework brings various benefits. On the one hand, the framework makes it possible for the user to select only those components that are necessary for him. On the other hand, this makes it easy to extend and modify the framework. For instance, ScaRLib, due to the actual need, has already integrated the `alchemist-scafi` module. If a user wants to use another simulator instead of Alchemist (e.g., FlameGPU [48] or VMAS [12]), he simply has to disable the import of this module and build a fresh custom environment based on his chosen simulator.

[1] `https://deeplearning4j.konduit.ai/`

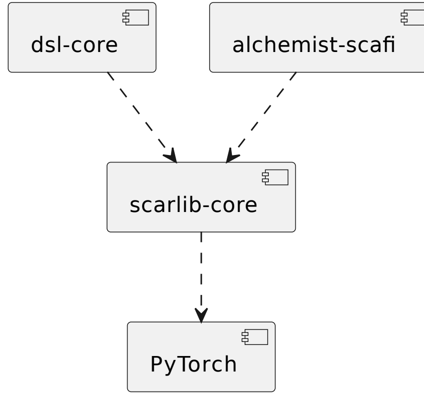


Figure 4.1: ScaRLib main modules.

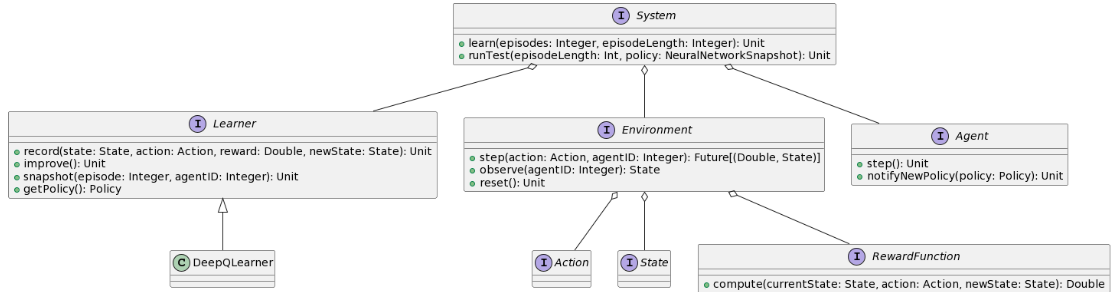
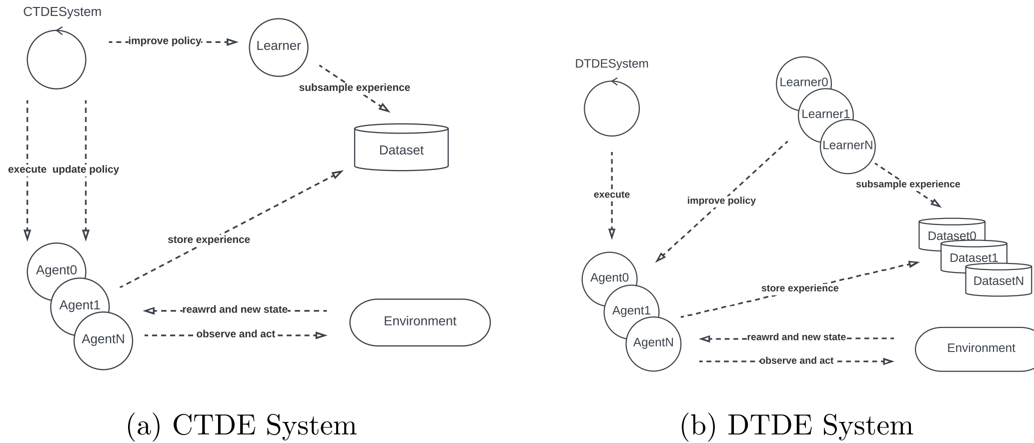


Figure 4.2: scarlib-core module UML class diagram.

ScaRLib Core

The `scarlib-core` module contains the essential abstractions of the reference domain, including the necessary data structures and learning algorithms. It has been conceptualised around several key components (Figure 4.2), the central one being the *system*. A system serves as a generic representation of a collection of agents interacting with each other and with a shared environment, trained to optimise a reward signal conveyed by a reward function. The tool offers two pre-implemented systems that are widely used in literature [25], namely the `CTDESystem` and `DTDESystem`. The distinction between them is based on the training of the different agents. To understand their dynamics better, it is useful to study their internal details. Both systems undergo a training process that includes a specific number of *epochs*, each of which comprises numerous *episodes*. Within a single episode, at a generic time step t, agents interact with the environment and receive the state s_t. They choose which action a_t to take based on this and the current policy π_t. The environment then gathers the actions of all agents and returns a new state s_{t+1} and the corresponding reward r_{t+1}. When using a



(a) CTDE System (b) DTDE System

Figure 4.3: 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.

`CTDESystem`, the agents are trained in a centralised way, involving a specialised agent, called the *Learner*, that gathers the experience of all the other agents to update the policy, and then the updated policy is distributed to all the agents, resulting in a *homogeneous* system. Conversely, when using a `DTDESystem`, each agent assumes responsibility for individual policy training, thus making the policies different from each other, resulting in a *heterogeneous* system. The behaviors of these systems are depicted in Figure 4.3.

Furthermore, the module provides a pre-implemented version of the DQN [40] learning algorithm, which can be used to train the agents. Alternatively, a user can define their own custom learning algorithm by creating a new class that extends the trait `Learner`.

DSL Core

The `dsl-core` module is dedicated to implementing a *Domain Specific Language* in Scala that can be used to define the high-level configuration of the experiment to run. This module is the simplest of the three, consisting of a series of *contextual functions* that make the definition of the learning system fluent and straightforward. The decision to implement a DSL, which essentially serves as a *facade* for the abstractions defined by the framework, was made to enable the identification of simple configuration errors at *compile time* rather than waiting for execution to intercept them.

Let's take the example of a user who wants to define his own learning system for the experiment he wants to run with the DSL. First of all, the basic components must be defined. To start with, a reward function must be defined.

```
class MyRewardFunction extends CollectiveRewardFunction:
    override def computeReward(state: State, action: Action,
       nextState: State): Double = ...
```

Afterward, it will be necessary to define the action space, which can be done as follows, leveraging Scala's *product types*:

```
sealed trait CustomAction extends Action
object CustomActionSpace:
    case object A extends CustomAction
    case object B extends CustomAction
    case object C extends CustomAction
    def all: Seq[CustomAction] = Seq(A, B, C)
```

Final refinements required include: i) choosing the class of the Alchemist environment to instantiate, ii) defining the number of agents living in the chosen environment, and iii) defining the size of the buffer in which the memory will be stored. Finally, everything can be combined to create the configuration of the learning system through the DSL as follows:

```
val system = learningSystem {
    rewardFunction { new MyRewardFunction() }
    actions { CustomActionSpace.all}
    dataset { ReplayBuffer[State, Action](10000) }
    agents { 50 } // select the number of agent
    environment {
        // select a specific environment
        "it.unibo.scarlib.experiments.myEnvironment"
    }
}
```

Alchemist-ScaFi

The `alchemist-scafi` (Figure 4.4) module has been designed to provide integration between the `scarlib-core` module and the two tools *Alchemist* [42] and *ScaFi* [18]. In addition to the `scarlib-core` module, this module provides an environment that implements bindings with the Alchemist simulator. This way, the user only needs to provide a YAML file containing the simulation specification

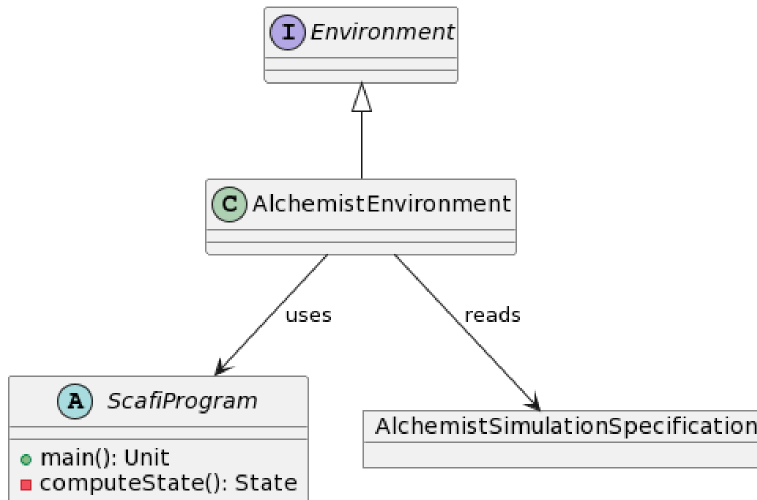


Figure 4.4: alchemist-scafi module UML class diagram.

they want to execute. Another component is the abstract class `ScafiProgram`, which represents the logic of the agents. In this case, the user only needs to implement the `computeState` method, specifying through *aggregate computing* how to reconstruct the state that will be returned to the various agents.

4.2 Component interactions

The *interaction* among the various components depends on the selected *training and execution model*. The framework provides two pre-implemented models, namely: `CTDESystem` and `DTDESystem`. The interaction flow for the CTDE model is illustrated in Figure 4.5, while that for the DTDE model is shown in Figure 4.6. In general, the main interaction takes place between the *agents* and the *environment*, in both models: initially, the agent observes the state of the environment and, based on this observation, makes a query to its policy to choose an action to execute. When the action is received, the environment computes the new state and returns a reward and the new state to the agent. At this point, the most significant difference between the two systems becomes evident. In the case of CTDE, the agent collects its experience in a centralized Learner that takes responsibility for updating the policy following the implemented learning algorithm. Conversely, in the case of DTDE, each agent interfaces directly with its own learner. In the former case, the policies of various agents will be homogeneous, whilst in the latter case, each agent will have its own policy.

An important aspect to note regarding figures 4.5 and 4.6 is that they represent a simplification of what actually happens. In fact, only one agent is illustrated,

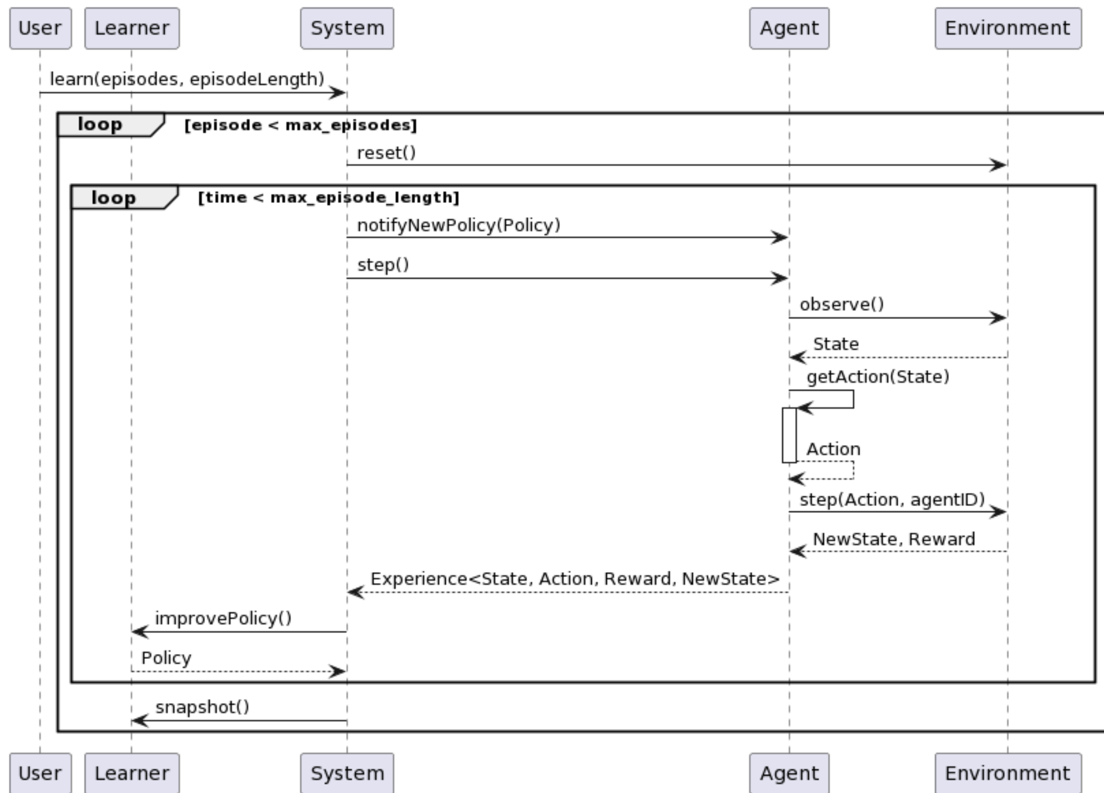


Figure 4.5: UML sequence diagram of the learning process using a CTDE system.

while in reality, the number of agents is much greater than 1. Both systems manage agents *concurrently*, leveraging asynchronous programming to enable parallel execution. The environment, in order to compute the new state, must wait to collect the actions of all agents for the current time step, thus following the model defined by the *markov games*[37].

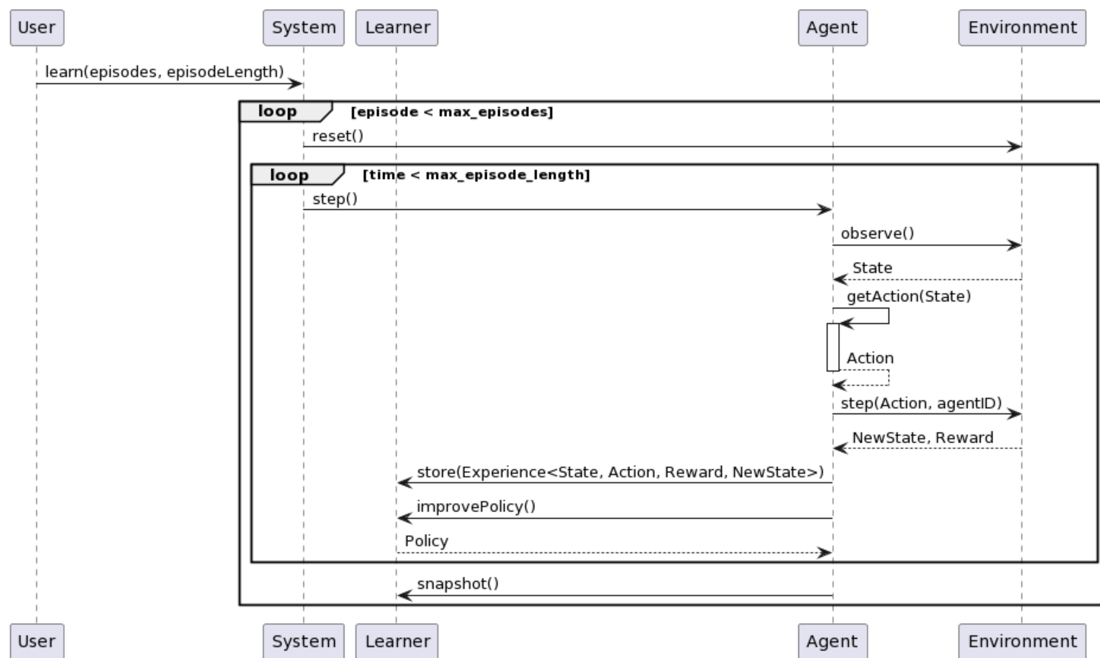


Figure 4.6: UML sequence diagram of the learning process using a DTDE system.

Chapter 5

Technologies

This chapter first presents the main technologies used to implement the framework, then it describes the process and devops techniques used to manage the development of the project.

5.1 Framework technologies

Scala language

Scala (short for *Scalable Language*) is a general-purpose programming language. It combines the *object-oriented* paradigm with the *functional* paradigm. The language is purely object-oriented in that *every value is an object*, while also incorporating the functional paradigm, ensuring that *every function is a value*. This permits the utilisation of higher-order functions, anonymous functions, and lambdas in a straightforward and natural manner, fostering programmers to implement pure functions. This implies that, for identical inputs, they returns the same output without side effects. This facilitates the implementation of more easily comprehensible and maintainable code with fewer errors.

The Scala language has been designed for the succinct, elegant, and type-safe expression of commons programming patterns. It is a *statically-typed* language, and due to the expressiveness of its type system, its abstractions can be used reliably and safely. Some of these abstractions include: upper and lower bounds for types, variance annotations, implicit parameters, implicit conversions, as well as polymorphic methods, among others. Moreover, a very important feature of the compiler is its highly potent *type inference*, which makes the code very readable, allowing programmers to avoid using unnecessary and redundant type annotations.

Originally, Scala was designed as a language intended for seamless interoperability with the JVM (Java Virtual Machine). Since then, native versions and

interoperability with JavaScript have also been included. The compiler does not directly manage support for cross-platform development; it is achieved by using plugins. These plugins enable the creation of an intermediate representation that contains cross-platform aspects, which is then used to generate the final code.

Asynchronous programming

Asynchronous programming is crucial within modern software development. The importance of asynchronous programming is its potential to improve the efficiency and responsiveness of applications. By permitting tasks to run concurrently without blocking the primary execution thread, it allows applications to handle multiple operations at once, making them more scalable and responsive to user interactions. In Scala, *Future* and *Promise* are crucial devices for handling asynchronous operations. A future denotes a value or error computation that may finish at some point in the future, allowing non-blocking execution. A promise, conversely, performs as a writable, synchronized container for a future result. This pairing encourages orderly and articulate code for dealing with asynchronous tasks, encouraging a more natural and manageable codebase.

ScalaPy

ScalaPy[1] is a software tool that effectively integrates Python and Scala, two prominent programming languages. This bridge empowers developers to utilise the strengths of both languages in a single project, offering flexibility and extensibility required for various tasks. At its heart, ScalaPy permits developers to utilise Python libraries and packages directly in Scala code, removing the requirement for challenging interoperability workarounds. This capability is notably beneficial when harnessing the extensive Python ecosystem, which comprises libraries for data science, machine learning, and scientific computing, such as NumPy, Pandas, and TensorFlow. By bridging this gap, ScalaPy enables Scala developers to tap into the rich functionality and pre-existing solutions offered by the Python community, thereby accelerating development and reducing duplication of effort. Moreover, ScalaPy provides robust support for data type conversions, allowing seamless passage of data between Python and Scala, further enhancing the interoperability between the two languages. This capability simplifies the process of combining Python's dynamic typing with Scala's strong, static typing, ensuring that the integration remains type-safe and reliable.

[1] `https://scalapy.dev/`

PyTorch

PyTorch [33] has become a groundbreaking framework for training *deep neural networks*, providing a potent and adaptable platform which has won over both researchers and developers in the field of deep learning. PyTorch's dynamic computation graph, sophisticated design, and user-friendly interface distinguish it from the rest, making it an indispensable tool for building and training neural networks. At the heart of PyTorch's appeal is its *dynamic computation graph*, a sharp contrast to the static graphs seen in many other deep learning frameworks. This dynamic nature allows developers to define and revise computational graphs as needed, empowering dynamic control flow and simplifying debugging. Researchers find this feature particularly useful in implementing complex models, encouraging experimentation and streamlining prototyping. PyTorch's capacity to harness *GPU acceleration* is effortless, allowing for the training of deep neural networks on *high-performance hardware*. *Autograd*, an automatic differentiation engine, effectively calculates gradients for optimization algorithms based on gradients such as *stochastic gradient descent* [5]. This characteristic streamlines the implementation of custom loss functions and intricate optimization strategies.

5.2 DevOps technologies used

DevOps techniques are fundamental to the development of a project for several reasons. On the one hand, they make it possible to improve code quality while keeping work organised, promoting testing and ensuring continuous integration of the various components being developed, while systematically tracking released versions. On the other hand, they make it easier to maintain high productivity by avoiding downtime and automating repetitive and tedious tasks that can be performed more efficiently by a computer than by a human, leaving more time for critical features.

Repository management

The management of the codebase was carried out using the renowned *Decentralized Version Control System (DVCS)* git [55], specifically leveraging the *GitHub* [2] hosting service. To ensure standardized and consistent tool usage, avoiding errors and compatibility issues, the *GitFlow* methodology was chosen. GitFlow is a branching model that involves the use of two main branches: `master` and `develop` (Figure 5.1). The `master` branch is used for *releases*, while the `develop` branch is used for *ongoing development*. Additionally, for each feature, a branch named

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

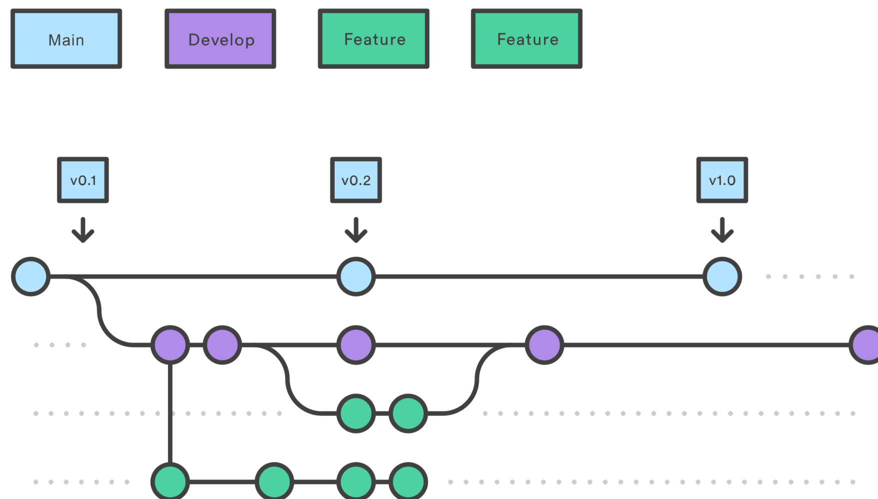


Figure 5.1: GitFlow.

`feature/feature-name` is created, which is then merged into the `develop` branch once the feature is completed. Furthermore, to ensure a standardized way of writing commit messages, the *Conventional Commits* [3] approach was adopted. A commit is therefore written in the following format: `<type>[<optional scope>]: <description>`. Some examples of commit types are `feat`, `fix`, `docs` and `style`.

Build automation

Build automation aims to automate the *management of dependencies*, *compilation*, and *distribution* of a software project. Firstly, build automation decreases human errors during source code compilation by eliminating potential sources of errors from manual omissions or configuration mistakes. This increases the software's reliability and accelerates the development cycle, enabling developers to concentrate on more creative tasks and problem-solving instead of manual compilation. Additionally, build automation aids in the regulation of *project dependencies* and *version control*, permitting developers to explicitly and systematically establish the necessary libraries and resources for the application and guaranteeing that they persistently maintain the accurate versions. Finally, build automation facilitates *continuous integration and continuous delivery (CI/CD)*, which are essential for timely releases of new features and bug fixes. Through automation, it is possible to implement *automated testing* and continuous deployment processes, decreasing release times and enhancing the overall quality of the software.

[3] `https://www.conventionalcommits.org/en/v1.0.0/`

In this project *Gradle* [4] was used as the build automation tool. It is a powerful and versatile tool extensively used in the software development industry. It supplies a flexible and efficient approach to regulate and automate different aspects of the build process for projects of all sizes and complexities. One of Gradle's notable advantages is its backing for multiple programming languages and platforms, which renders it a preferred choice for both Java-based applications and a wide variety of other technologies. With Gradle, developers are able to establish and adapt build scripts by utilising a Kotlin-based DSL (Domain Specific Language), providing considerable versatility and expressiveness. Gradle also performs exceedingly well in dependency management, allowing developers to simply declare and manage project dependencies, making it an essential tool for building and maintaining strong software projects. Whether building, testing or deploying applications, Gradle streamlines and automates these tasks to enhance productivity and allow developers to concentrate on writing superior code. Most of the Gradle configuration is in the `build.gradle.kts` file. For example, it's possible to specify a dependency on an third-party library as shown below:

```
dependencies {
    implementation("it.unibo.alchemist:alchemist-incarnation-scafi
       :25.14.6")
}
```

Another crucial aspect is the ability to define custom tasks, allowing for a series of actions to be carried out during the build stage. For instance, the following task creates a jar file that holds the documentation of the program's code.

```
val scaladocJar by tasks.registering(Jar::class) {
    dependsOn("scaladoc")
    val destinationDirectory = tasks
            .named<ScalaDoc>("scaladoc")
            .get()
            .destinationDir
    from(destinationDirectory)
    archiveClassifier.set("docs-${project.name}")
}
```

Continuous integration

Continuous Integration (CI) is a software development technique that involves regularly integrating code modifications into a shared repository following automated

[4] `https://gradle.org/`

building and testing procedures. Its significance lies in its ability to optimise the development workflow by pinpointing and resolving problems early on in the development cycle. CI ensures that modifications made by multiple developers do not cause any conflicts or bugs. This improves code quality, reduces integration issues, and ultimately speeds up the software delivery. By automating these processes, CI not only saves time but also encourages collaboration and motivates developers to write reliable and maintainable code. This results in a more efficient and robust software development pipeline.

In this project *GitHub Actions* [5] were used to implement CI. GitHub Actions is a *CI/CD service*, integrated into GitHub, that allows developers to automate their software development workflows. GitHub Actions is based on the concept of *workflows*, which are a series of jobs that are executed when a specific event occurs. For example, a workflow can be triggered when a pull request is opened or when a commit is pushed to the repository. Workflows are defined in a YAML file called `.github/workflows/main.yml`. The following is an example of a workflow that is triggered when a commit is pushed to the `main` branch. It is composed of one job named `build` that runs on a `ubuntu` machine. The job consists of four steps: first it checks out the code, then it sets up the Node.js environment, then it installs project dependencies, and finally it runs the tests.

```
1  name: Simple Pipeline
2
3  on:
4   push:
5    branches:
6     - main
7
8  jobs:
9   build:
10    runs-on: ubuntu-latest
11
12    steps:
13    - name: Checkout Code
14     uses: actions/checkout@v2
15
16    - name: Set up Node.js
17     uses: actions/setup-node@v2
18     with:
19      node-version: '14'
20
```

[5] https://docs.github.com/en/actions

```
21     - name: Install Dependencies
22       run: npm install
23
24     - name: Run Tests
25       run: npm test
```

Versioning and releasing

Software versioning refers to the process of assigning a unique identifier to a software state. The identifier is typically an alphanumeric sequence of characters separated by dots, slashes, or dashes. In this project, to distinguish different versions of the software, we decided to follow the guidelines proposed by *Semantic Versioning* [6]. Consequently, the software version is represented by three numbers separated by a dot: `MAJOR.MINOR.PATCH`.

The correct version to be associated with the software state is based on the saved commits, which were written following the Conventional Commit approach. In particular, we used the following approach:

- **MAJOR** release: any commit type and scope that causes a breaking change;
- **MINOR** release: any commit with type `feat`;
- **PATCH** release: any commit with type `fix`, `doc`, `perf`, `revert`.

With regard to the *release* of the software, on the other hand, it was decided to publish the various versions on the *Maven Central repository* [7], so as to make the framework easily available and importable for all users. The release process, following good devops practices, was also fully automated.

5.3 License

A *software license* is a legal agreement that governs the terms and conditions under which a user may use a particular piece of software. It serves as a critical tool in the world of software development and distribution because it defines the rights and responsibilities of both the software creator (licensor) and the end user (licensee). The importance of a software licence lies in several key aspects. First, it helps protect the software developer's intellectual property rights by specifying how the software can be used, copied, modified and distributed. This ensures that

[6] https://semver.org/
[7] https://central.sonatype.com/

the developer retains control over his or her creation and can potentially generate revenue from it. Secondly, a well-drafted licence can provide legal protection for both parties by clarifying liability and warranty terms, thereby reducing the risk of disputes and litigation. Finally, software licences can promote responsible and ethical use of software by preventing piracy and unauthorised distribution, which ultimately benefits the software industry as a whole and encourages innovation. In summary, software licences are essential because they provide a framework for fair, legal and mutually beneficial interactions in the software ecosystem.

ScaRLib license ScaRLib is distributed under the *MIT license*. The *Massachusetts Institute of Technology License*, commonly referred to as the MIT License, is an open-source software license that has gained widespread use. Its simplicity and permissiveness enable developers to utilise, adjust, distribute, and even commercialise the software without significant restrictions. Users are usually expected to include the original copyright notice and disclaimer when redistributing the software. This licence advocates *collaboration* and *code sharing* within the *open-source community* while ensuring legal protection for creators and users. Its straightforward terms make it a popular choice for developers intending to utilise or contribute to open-source projects.

Chapter 6

Validation

To test the functionalities of the framework, a series of experiments were created[1] using selected features from well-known problems in literature. This allowed for a large number of agents to be involved and non-trivial coordination tasks to be undertaken.

6.1 Cohesion and collision

Description The aim of this experiment is to create a flock of drones with the task of avoiding *collisions* and maintaining a *cohesive* movement pattern. The goal is to learn a policy that guides each agent's movement based on the distances from neighbors. This problem is well-known in the literature as *flocking* [47, 72].

In our study, we examine an unbounded 2D environment where every agent has a set number of neighbours (the five nearest, a hyper-parameter that is configurable), and can move in eight directions (the four cardinal points and the four diagonals). The state of the environment is reconstructed through aggregate computing, using ScaFi, as follows:

```
val state = foldhoodPlus(Seq.empty)(_ ++ _)(Set(nbrVector))
```

In the code above: i) `nbrVector` represents relative distances to neighbours, ii) `foldhoodPlus` is a ScaFi function that iterates over all neighbours, and iii) `++` is the concatenation operation between sequences.

A crucial aspect of this task involves defining the *reward function*. The aim is to learn a policy that enables agents, initially placed randomly within the environment, to approach one another and reach a specific *target distance* δ (set in advance as a parameter of the experiment) without any collisions occurring.

[1] https://github.com/ScaRLib-group/ScaRLib-flock-demo

Figure 6.1: 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.

In this case, a function composed of two components was selected. The *collision factor* kicks in when the distance is less than the target distance:

$$\text{collision} = \begin{cases} 0 & \text{if } d > \delta \\ \exp\left(-\frac{d}{\delta}\right) & \text{otherwise} \end{cases} \tag{6.1}$$

Following this function, the distance d from the nearest neighbour is *exponentially weighted*, causing agents to move away from each other. The second element aims to enhance *cohesion*. Considering the neighbour with the longest distance D, the reward function is defined as follows:

$$\text{cohesion} = \begin{cases} 0 & \text{if } d < \delta \\ -(D - \delta) & \text{otherwise} \end{cases} \tag{6.2}$$

The overall reward function is defined as the sum of these two factors ($cohesion + collision$) as shown in Figure 6.1.

Implementation First of all, it was necessary to define the action space of the various agents, this can be done easily by leveraging the product types of Scala as follows:

```
import it.unibo.scarlib.core.model.Action

object CohesionCollisionActionSpace {
    final case object North extends Action
    final case object South extends Action
```

```
    final case object East extends Action
    final case object West extends Action
    final case object NorthEast extends Action
    final case object NorthWest extends Action
    final case object SouthWest extends Action
    final case object SouthEast extends Action

    def all(): Seq[Action] =
        Seq(North, South, East, West, NorthEast, NorthWest,
           SouthWest, SouthEast)
}
```

Secondly, it was necessary to define the state space:

```
import it.unibo.scarlib.core.model.State
import StateInfo.{neighborhood, encoding}

case class CohesionCollisionState (
    positions: List[(Double, Double)],
    agentId: Int
) extends State {

    override def elements(): Int = neighborhood * encoding

    override def toSeq(): Seq[Double] = {
        val fill = List.fill(elements())(0.0)
        (positions
            .flatMap { case (l, r) => List(l, r) } ++ fill)
            .take(elements())
    }

    override def isEmpty(): Boolean = false
}
```

Then, it was necessary to define the reward function:

```
import it.unibo.scarlib.core.model.{Action, RewardFunction, State}
import it.unibo.scarlib.core.util.AgentGlobalStore

class CohesionCollisionRewardFunction extends RewardFunction {
   private val targetDistance = 0.2

```

```
    override def compute(currentState: State, action: Action,
        nextState: State): Double = {
        val distances = computeDistancesFromNeighborhood(s)
        val cohesion = cohesionFactor(distances)
        val collision = collisionFactor(distances)
        AgentGlobalStore()
            .put(s.agentId, "cohesion", cohesion)
        AgentGlobalStore()
            .put(s.agentId, "collision", collision)
        AgentGlobalStore()
            .put(s.agentId, "reward", collision + cohesion)
        cohesion + collision
    }

    private def cohesionFactor(distances: Seq[Double]): Double = {
        val max = distances.max
        if (max < targetDistance){
            0.0
        } else {
            -(max - targetDistance)
        }
    }

    private def collissionFactor(distances: Seq[Double]): Double = {
        val min: Double = distances.min
        if (min < targetDistance) {
            2 * math.log(min / targetDistance)
        } else {
            0.0
        }
    }
}
```

Finally, it was necessary to define the aggregate program used to reconstruct the state of each agent:

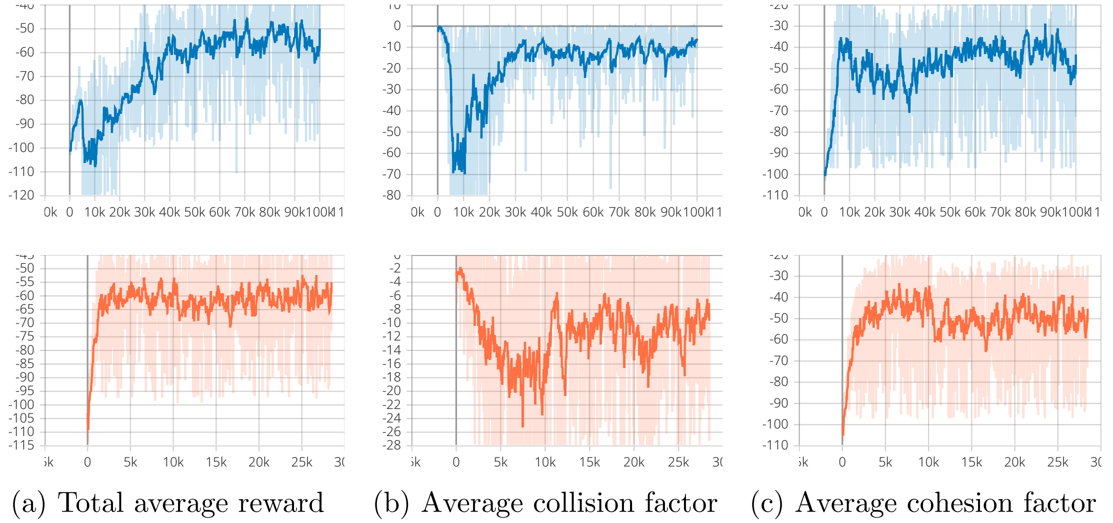
```

import it.unibo.scafi.ScafiProgram

class CohesionCollisionScafiAgent
    extends ScafiProgram with FieldUtils {
```

```
6     override protected def computeState(): State = {
7         val positions = excludingSelf
8             .reifyField(nbrVector())
9             .toList
10            .sortBy(_._2.distance(Point3D.Zero))
11            .map(_._2)
12            .map(point => (point.x, point.y))
13            .take(5)
14        CohesionCollisionState(positions, mid())
15    }
16 }
```

Results The experiment's training involved conducting 1000 *epochs*, each with 100 *episodes*, using 50 *agents*, an *environment* measuring 50x50 metres and a *target distance* δ set to 2 metres. The training was conducted using both *CTDE* and *DTDE* processes.

Figure 6.2 shows the *multi-objective* nature of the problem. In fact, cohesion and collision are two *adverse signals*, and the system had to find a *balance* between these two values. The graphs show that the learning algorithm optimizes one signal at a time, with cohesion tending towards zero and collision increasing. Nonetheless, after 500 epochs in CTDE simulation, it is possible to see that the system had already found a balance between these two factors. Instead, in the case of DTDE learning, convergence can be observed after approximately 50 epochs, which is due to the presence of a larger number of policies.

To verify the *homogenous policy* learned using the CTDE process, 16 simulations were conducted, each with agents randomly positioned and varying the seeds. Given the homogenous nature of the policy learned with CTDE, we also varied the *number of agents* by conducting simulations with 50, 100 and 200 agents. According to our *hypotheses*, this should not impact the *quality* of the learned policy and its *performance*. Figure 6.3 displays screenshots from a simulation with 50 and 200 agents, illustrating how they gradually form *cohesive clusters* over time. Figure 6.4 shows the performance of the learned policy with 50, 100 and 200 agents. From these graphs, it can be observed that the *performance does not change significantly* with varying numbers of agents, and the system is able to maintain approximately a distance δ between the agents.



(a) Total average reward (b) Average collision factor (c) Average cohesion factor

Figure 6.2: Cohesion and collision experiment results. The y-axis represents the reward value. The x-axis represents the total number of timesteps. The first three graphs show the results of the CDTE learning process, while the last three show the results of the DTDE learning process.

6.2 Follow the leader

Description The aim of this experiment is to create a flock of drones with the task of avoiding *collissions* while following a special agent, the *leader*. The goal is to learn a policy that guides each agent's movement based on the distances from neighbors and the leader.

In our study, we examine an unbounded 2D environment where each agent has a set number of neighbours (the five nearest, a hyper-parameter that is configurable), and can move in eight directions (the four cardinal points and the four diagonals). During training the leader is placed in a random position and cannot move, instead, during evaluation, the leader choose a random direction every h time steps (a hyper-parameter that is configurable).

The *reward function* is composed of two components. The first component is the *collision factor*, which is the same as the one used in the previous experiment, kicks in when the distance is less than a given target distance δ:

$$\text{collision} = \begin{cases} 0 & \text{if } d > \delta \\ \exp\left(-\frac{d}{\delta}\right) & \text{otherwise} \end{cases} \tag{6.3}$$

The second component aims to reduce the distance between each agent and the leader:

$$\text{distance to leader} = -d \tag{6.4}$$

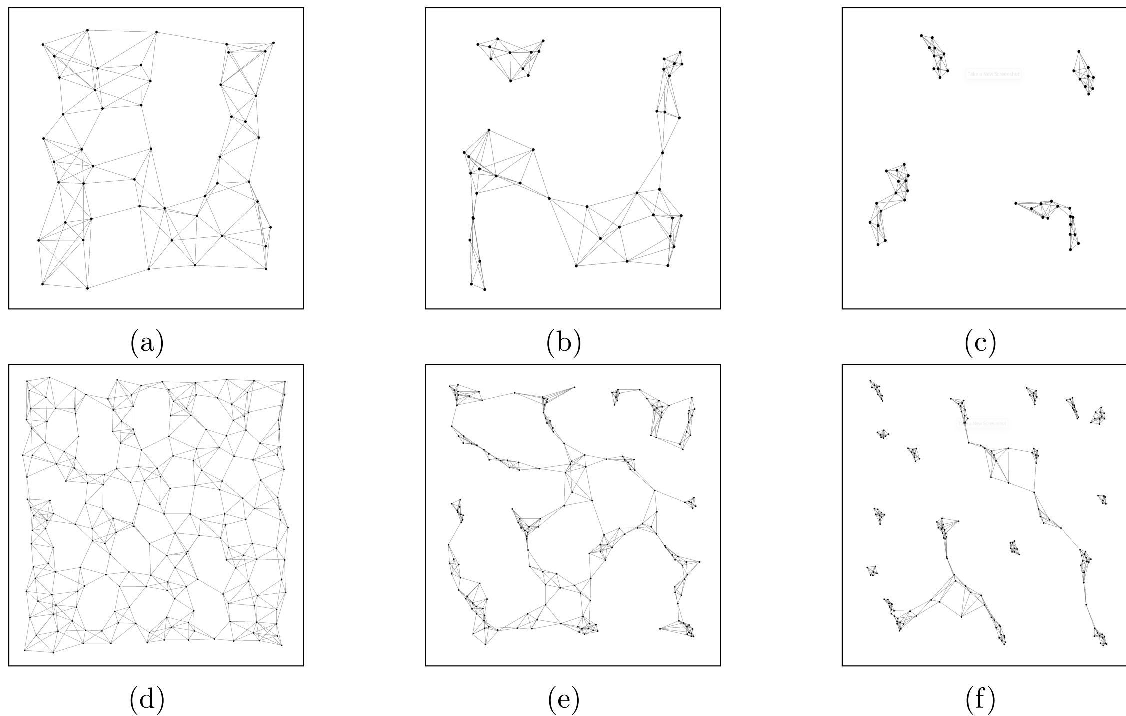
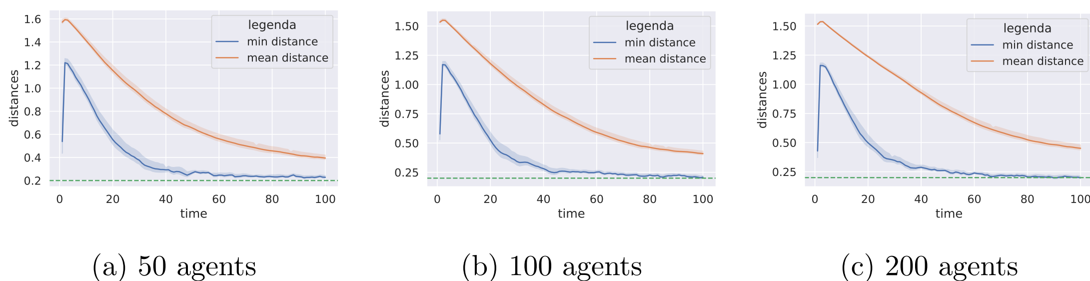


Figure 6.3: Snapshots of the learned policy, the time flow is from left to right. In the first row, there are 50 agents, whereas in the second row, there are 200 agents. In the last step of the simulation, the agents converged to a distance of approximately δ.



(a) 50 agents (b) 100 agents (c) 200 agents

Figure 6.4: 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).

The overall reward function is defined as the sum of these two factors.

Implementation First of all, it was necessary to define the action space of the various agents, this can be done easily by leveraging the product types of Scala as follows:

```
import it.unibo.scarlib.core.model.Action

object FollowTheLeaderActionSpace {
    final case object North extends Action
    final case object South extends Action
    final case object East extends Action
    final case object West extends Action
    final case object NorthEast extends Action
    final case object NorthWest extends Action
    final case object SouthWest extends Action
    final case object SouthEast extends Action
    final case object StandStill extends Action

    def all(): Seq[Action] =
        Seq(North, South, East, West, NorthEast, NorthWest,
            SouthWest, SouthEast, StandStill)

    def sample(): Action =
        Random.shuffle(all().take(all().size - 1)).head
}
```

Secondly, it was necessary to define the state space:

```
import it.unibo.scarlib.core.model.State
import StateInfo.{neighborhood, encoding}


case class FollowTheLeaderState (
    directionToLeader: (Double, Double),
    positions: List[(Double, Double)],
    distanceFromLeader: Double,
    agentId: Int
) extends State {

    override def elements(): Int = neighborhood * encoding

```

```
14     override def toSeq(): Seq[Double] = {
15         val fill = List.fill(elements())(0.0)
16         (positions
17             .flatMap { case (l, r) => List(l, r) }
18             ++ List(directionToLeader._1, directionToLeader._2)
19             ++ fill)
20             .take(elements())
21     }
22 
23     override def isEmpty(): Boolean = false
24 }
```

Then, it was necessary to define the reward function:

```
1  import it.unibo.scarlib.core.model.{Action, RewardFunction, State}
2  import it.unibo.scarlib.core.util.AgentGlobalStore
3  
4  class FollowTheLeaderRewardFunction extends RewardFunction {
5  
6      private val targetDistance = 0.2
7  
8      override def compute(currentState: State, action: Action,
          nextState: State): Double = {
9          val distances = computeDistancesFromNeighborhood(s)
10         val collision = collisionFactor(distances)
11         val distanceFactor = - s.distanceFromLeader
12         AgentGlobalStore()
13             .put(s.agentId, "distance", distanceFactor)
14         AgentGlobalStore()
15             .put(s.agentId, "collision", collision)
16         AgentGlobalStore()
17             .put(s.agentId, "reward", collision + distanceFactor)
18         distanceFactor + collision
19     }
20 
21     private def collissionFactor(distances: Seq[Double]): Double =
          {
22         val min: Double = distances.min
23         if (min < targetDistance) {
24             2 * math.log(min / targetDistance)
25         } else {
```

```
            0.0
        }
    }
}
```

Finally, it was necessary to define the aggregate program used to reconstruct the state of each agent:

```

import it.unibo.scafi.ScafiProgram

class CohesionCollisionScafiAgent
    extends ScafiProgram with FieldUtils {

    private val leader = sense[Int]("leaderId") == mid()

    override protected def computeState(): State = {
        val potentialToLeader = classicGradient(leader, nbrRange)
        val nearestToLeader = includingSelf
            .reifyField((nbr(potentialToLeader), nbrVector()))
            .minBy(_._2._1)._2._2
        val positions = excludingSelf
            .reifyField(nbrVector())
            .toList
            .sortBy(_._2.distance(Point3D.Zero))
            .map(_._2)
            .map(point => (point.x, point.y))
            .take(5)
        val leaderNode = alchemistEnvironment
            .getNodes
            .get(leaderId)
        val myself = alchemistEnvironment
            .getNodes
            .get(mid())
        val distance = alchemistEnvironment
            .getDistanceBetweenNodes(myself, leaderNode)

        FollowTheLeaderState(
            (nearestToLeader.x, nearestToLeader.y),
            positions,
            distance,
```

```
34                 mid()
35             )
36     }
37 }
```

Results The experiment's training involved conducting 1000 *epochs*, each with 100 *episodes*, using 50 *agents*, an *environment* measuring 50x50 metres and a *target distance* δ set to 2 metres. The training was conducted using a *CTDE* process.

Figure 6.5 shows the results of the training. The graphs show that the learning algorithm optimizes one signal at a time, with distance to leader tending towards zero and collision increasing. Nonetheless, after some epochs, it is possible to see that the system had already found a balance between these two factors.

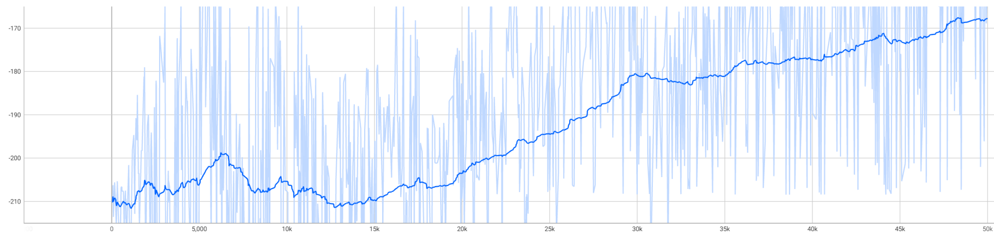
To verify the *homogeneous policy* learned using the CTDE process, 16 simulations were conducted, each with agents randomly positioned and varying the seeds. Given the homogeneous nature of the policy learned with CTDE, we also varied the *number of agents* by conducting simulations with 50, 100 and 200 agents. According to our *hypotheses*, this should not impact the *quality* of the learned policy and its *performance*. Figure 6.6 shows screenshots from a simulation with 50 agents, illustrating how they gradually tend to get closer to the leader (who is represented by the agent with the blue circle). Figure 6.7 shows the performance of the learned policy with 50, 100 and 200 agents. From these graphs, it can be observed that the *performance does not change significantly* with varying numbers of agents.

6.3 Discussion

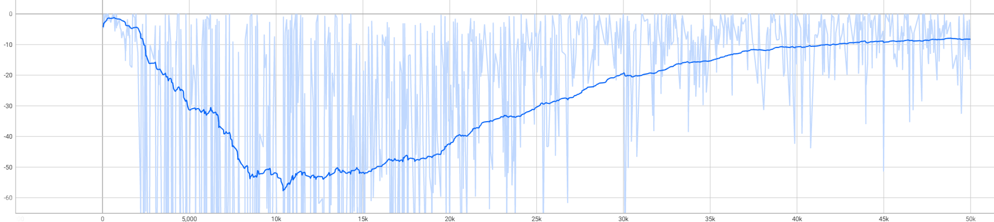
This section discusses the current state of the framework, highlighting its strengths and weaknesses that will shape future work in Chapter 7.

Performance Currently, a thorough and comprehensive performance benchmarking has not been carried out. This choice was made to favor a simpler, more modular, and flexible design and implementation, following the principle of: "Avoid premature optimization. First make it right, then make it fast" [32]. Nonetheless, this aspect is crucial and will undoubtedly be considered in future work to ensure the tool is fully prepared for community use.

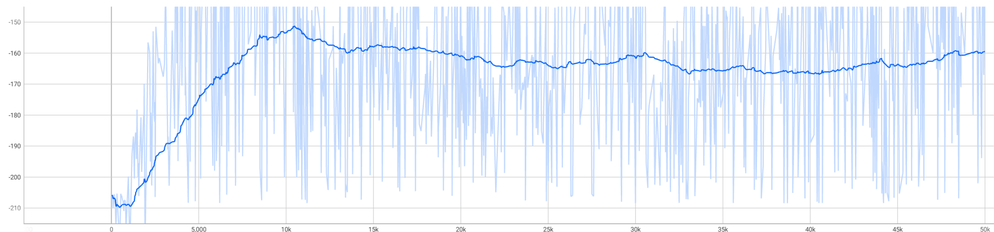
Training and execution model Currently, the framework offers two training and execution models, both among the most renowned in literature. While these



(a) Total average reward



(b) Average collision factor

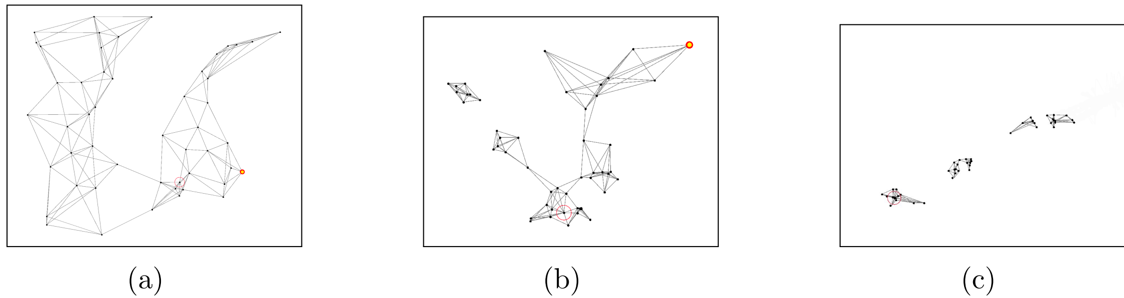


(c) Average distance to leader factor

Figure 6.5: Follow the leader training experiment results. The y-axis represents the reward value. The x-axis represents the total number of timesteps.

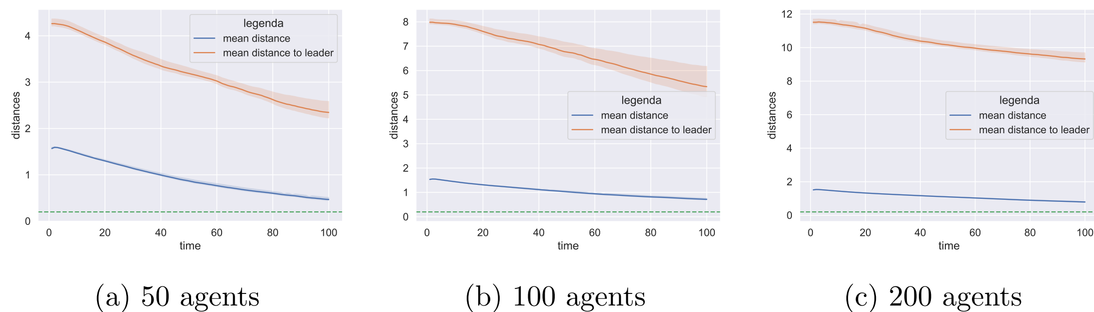
models can address various problems, they are not exclusive and may limit functionality in certain scenarios. To facilitate user implementation and avoid the need to start from scratch, the framework must be expanded to include additional models.

Scala-Python integration The integration between ScaRLib, which is based on the Scala language, and the deep learning framework PyTorch, which is based on Python, is facilitated by the ScalaPy tool. However, despite the usefulness and potency of this tool, it remains in an experimental and research phase that imposes certain usability and configuration restrictions, particularly in non-Linux environments.



(a) (b) (c)

Figure 6.6: Snapshots of the learned policy, the time flow is from left to right.



(a) 50 agents (b) 100 agents (c) 200 agents

Figure 6.7: 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. The blue line represents the average distance between the agents, while the orange line represents the average distance between each agent and the leader.

Chapter 7

Conclusions

The work conducted in this thesis has resulted in the creation of *ScaRLib*. The primary objective of this framework is to minimize complexity in the design and building of *multi-agent cyber-physical systems* (such as swarms of robots, IoT sensor networks, and various others) exploiting and merging the potential of the two established methods employed in developing these systems, namely: *macro-programming* (i.e., aggregate computing) and *artificial intelligence* (i.e., reinforcement learning).

In order to achieve this goal, the framework has been designed in a modular and extensible way, which has two main advantages. Firstly, users can effortlessly integrate fresh components including distinct simulators, various learning engines, and new learning algorithms or models. Secondly, users can leverage solely the elements of the framework that are required for their specific project needs, without being duty-bound to use all of the modules.

A key aspect was to create abstractions within the reference domain to allow a concise and unambiguous definition of a new experiment. Furthermore, standard training and execution models were pre-implemented (i.e., *CTDE* and *DTDE*), alongside a well-known learning algorithm (i,e., *DQN*). Afterwards, the core framework was combined with two tools, *Alchemist* and *ScaFi*. Alchemist permits the simulation of highly complex distributed systems whereas ScaFi enables the functional use of the aggregate computing paradigm through a Scala programming language API. Additionally, a *Domain Specific Language* was developed to declaratively specify experiments and preemptively detect configuration errors during compile time, avoiding the need to wait for runtime occurrences.

In our opinion, the framework serves as an excellent starting point for developing such systems with a hybrid method and streamlines the definition of complex new experiments. Our aim is to showcase the potential of this technique and inspire the scientific community to delve further into this realm.

7.1 Future work

Despite the work that has been done, the reference domain is very broad, and many aspects still require further improvement. This section aims to outline the primary areas that need to be explored.

Performance The performance of the framework has only been assessed in relation to a small number of moderately challenging use cases. However, it is essential to conduct further research in order to detect potential bottlenecks and ensure that the minimum desired level of performance can be achieved, thus meeting the diverse needs of users.

Execution model Currently, ScaRLib has pre-implemented integration with the Alchemist simulator via a basic, single execution model. This model operates through each agent interacting with the environment at each timestep, followed by the environment awaiting action collection before transition to a new state. While this configuration is typical in this field, it will be necessary to explore new solutions that enable the development of more specialised systems.

Learning framework The framework currently supports solely the PyTorch learning framework. However, it would be beneficial to include additional frameworks, such as TensorFlow [1] or DL4J, to provide users with a wider range of options.

Scala-Pyhton integration Currently, the framework utilises the ScalaPy tool to integrate Scala and Python programming languages. Despite its potential, the tool has inherent research tool limitations, necessitating a judicious evaluation of continuing its use or exploring alternative options, including custom bindings or third-party tools.

Online learning The inclusion of *online learning* models in the framework, namely models that can continue learning after deployment, would provide significant value. However, this is a field with few existing solutions, made more complicated by the domain's multi-agent nature. We suggest leveraging *federated learning* [71, 2] as an interesting approach, enabling agents to exchange parts of their policy neural networks. This approach may have the potential to *distribute knowledge* within the system. Nevertheless, additional research is required, and alternative options should also be taken into account.

Bibliography

[1] Martín Abadi. Tensorflow: learning functions at scale. In *Proceedings of the 21st ACM SIGPLAN international conference on functional programming*, pages 1–1, 2016.

[2] Sawsan AbdulRahman, Hanine Tout, Hakima Ould-Slimane, Azzam Mourad, Chamseddine Talhi, and Mohsen Guizani. A survey on federated learning: The journey from centralized to distributed on-site learning and beyond. *IEEE Internet of Things Journal*, 8(7):5476–5497, 2020.

[3] Gregory D Abowd. Beyond weiser: From ubiquitous to collective computing. *Computer*, 49(1):17–23, 2016.

[4] Gianluca Aguzzi, Roberto Casadei, Danilo Pianini, and Mirko Viroli. Dynamic decentralization domains for the internet of things. *IEEE Internet Computing*, 26(6):16–23, nov 2022.

[5] Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185–196, 1993.

[6] Aykut Argun, Agnese Callegari, and Giovanni Volpe. *Simulation of Complex Systems*. IOP Publishing, 2021.

[7] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. 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*, pages 1–20. Springer, 2018.

[8] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu-an Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. Parsec: A parallel simulation environment for complex systems. *Computer*, 31(10):77–85, 1998.

[9] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula, 2019.

[10] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *Computer*, 48(9):22–30, 2015.

[11] Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. Self-adaptation to device distribution in the internet of things. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 12(3):1–29, 2017.

[12] Matteo Bettini, Ryan Kortvelesy, Jan Blumenkamp, and Amanda Prorok. Vmas: a vectorized multi-agent simulator for collective robot learning. *arXiv preprint arXiv:2207.03530*, 2022.

[13] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm intelligence: from natural to artificial systems.* Number 1. Oxford university press, 1999.

[14] Alan H Bond and Les Gasser. *Readings in distributed artificial intelligence.* Morgan Kaufmann, 2014.

[15] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7:1–41, 2013.

[16] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, 2008.

[17] Margaret Roseann Cannon-Diehl. Simulation in healthcare and nursing: State of the science. *Critical care nursing quarterly*, 32(2):128–136, 2009.

[18] Roberto Casadei, Mirko Viroli, Gianluca Aguzzi, and Danilo Pianini. Scafi: A scala dsl and toolkit for aggregate programming. *SoftwareX*, 20:101248, 2022.

[19] Roberto Casadei, Mirko Viroli, Giorgio Audrito, Danilo Pianini, and Ferruccio Damiani. Engineering collective intelligence at the edge with aggregate processes. *Engineering Applications of Artificial Intelligence*, 97:104081, January 2021.

[20] HeeSun Choi, Cindy Crump, Christian Duriez, Asher Elmquist, Gregory Hager, David Han, Frank Hearl, Jessica Hodgins, Abhinandan Jain, Frederick

Leve, et al. On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward. *Proceedings of the National Academy of Sciences*, 118(1):e1907856118, 2021.

[21] Felipe Leno Da Silva, Matthew E Taylor, and Anna Helena Reali Costa. Autonomously reusing knowledge in multiagent reinforcement learning. In *IJCAI*, pages 5487–5493, 2018.

[22] S Dolev. Self-stabilization, march 2000, 2000.

[23] Davide Domini, Filippo Cavallari, Gianluca Aguzzi, and Mirko Viroli. Scarlib: A framework for cooperative many agent deep reinforcement learning in scala. In Sung-Shik Jongmans and Antónia Lopes, editors, *Coordination Models and Languages*, pages 52–70, Cham, 2023. Springer Nature Switzerland.

[24] Ali Dorri, Salil S Kanhere, and Raja Jurdak. Multi-agent systems: A survey. *Ieee Access*, 6:28573–28593, 2018.

[25] Wei Du and Shifei Ding. A survey on multi-agent deep reinforcement learning: from the perspective of challenges and applications. *Artificial Intelligence Review*, 54(5):3215–3238, November 2020.

[26] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[27] Giancarlo Fortino, Raffaele Gravina, and Stefano Galzarano. *Wearable computing: From modeling to implementation of wearable systems based on body sensor networks*. John Wiley & Sons, 2018.

[28] Frédérick Garcia and Emmanuel Rachelson. Markov decision processes. *Markov Decision Processes in Artificial Intelligence*, pages 1–38, 2013.

[29] Pablo Hernandez-Leal, Michael Kaisers, Tim Baarslag, and Enrique Munoz De Cote. A survey of learning in multiagent environments: Dealing with non-stationarity. *arXiv preprint arXiv:1707.09183*, 2017.

[30] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. Is multiagent deep reinforcement learning the answer or the question? A brief survey. *CoRR*, abs/1810.05587, 2018.

[31] John Hunt. Introduction to akka actors. In *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, pages 383–398. Springer, 2014.

[32] Randall Hyde. The fallacy of premature optimization, 2006.

[33] Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. Pytorch. *Programming with TensorFlow: Solution for Edge Computing Applications*, pages 87–104, 2021.

[34] Mohammad Jafari and Hao Xu. Biologically-inspired intelligent flocking control for networked multi-uas with uncertain network imperfections. *Drones*, 2(4), 2018.

[35] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *J. Artif. Int. Res.*, 4(1):237–285, may 1996.

[36] 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*, pages 101–116. Springer, 2015.

[37] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In William W. Cohen and Haym Hirsh, editors, *Machine Learning Proceedings 1994*, pages 157–163. Morgan Kaufmann, San Francisco (CA), 1994.

[38] M. Mamei, F. Zambonelli, and L. Leonardi. Cofields: a physically inspired approach to motion coordination. *IEEE Pervasive Computing*, 3(2):52–61, 2004.

[39] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. Softw. Eng. Methodol.*, 18(4), jul 2009.

[40] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[41] Manish Parashar and Salim Hariri. Autonomic computing: An overview. In *International workshop on unconventional programming paradigms*, pages 257–269. Springer, 2004.

[42] D Pianini, S Montagna, and M Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. *Journal of Simulation*, 7(3):202–215, August 2013.

[43] Danilo Pianini, Roberto Casadei, and Mirko Viroli. Self-stabilising priority-based multi-leader election and network partitioning. In *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 81–90. IEEE, 2022.

[44] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7(3):202–215, 2013.

[45] Carlo Pinciroli and Giovanni Beltrame. Buzz: A programming language for robot swarms. *IEEE Software*, 33(4):97–100, 2016.

[46] Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

[47] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In Maureen C. Stone, editor, *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987, Anaheim, California, USA, July 27-31, 1987*, pages 25–34. ACM, 1987.

[48] 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, page 1125–1126, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[49] Julien Roy, Paul Barde, Félix Harvey, Derek Nowrouzezahrai, and Chris Pal. Promoting coordination through policy regularization in multi-agent deep reinforcement learning. *Advances in Neural Information Processing Systems*, 33:15774–15785, 2020.

[50] Sangita Roy, Samir Biswas, and Sheli Sinha Chaudhuri. Nature-inspired swarm intelligence and its applications. *International Journal of Modern Education and Computer Science*, 6(12):55, 2014.

[51] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philip H. S. Torr, Jakob Foerster, and Shimon Whiteson. The starcraft multi-agent challenge, 2019.

[52] Hartmut Schmeck. Organic computing–a generic approach to controlled self-organization in adaptive systems. In *Multiagent System Technologies: 9th*

German Conference, MATES 2011, Berlin, Germany, October 6-7, 2011. Proceedings 9, pages 2–2. Springer, 2011.

[53] Melanie Schranz, Gianni A Di Caro, Thomas Schmickl, Wilfried Elmenreich, Farshad Arvin, Ahmet Şekercioğlu, and Micha Sende. Swarm intelligence and cyber-physical systems: concepts, challenges and future trends. *Swarm and Evolutionary Computation*, 60:100762, 2021.

[54] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[55] Diomidis Spinellis. Git. *IEEE software*, 29(3):100–101, 2012.

[56] Kefan Su and Zongqing Lu. Divergence-regularized multi-agent actor-critic. In *International Conference on Machine Learning*, pages 20580–20603. PMLR, 2022.

[57] Y Tan. Swarm robotics: collective behavior inspired by nature. *J Comput Sci Syst Biol*, 6(2013):e106, 2013.

[58] Kagan Tumer and David Wolpert. A survey of collectives. In *Collectives and the design of complex systems*, pages 1–42. Springer, 2004.

[59] Alan M Turing. *Computing machinery and intelligence*. Springer, 2009.

[60] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[61] Banu Calis Uslu and Multi-Agent Agent. The role of mas interoperability for iot applications: A survey on recent advances in manufacturing systems. *Journal of the Faculty of Engineering and Architecture of Gazi University*, 38(2):1279–1297, 2023.

[62] Mirko Viroli. Programming very-large scale systems of wearables. 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*, UbiComp/ISWC'15 Adjunct, page 887–888, New York, NY, USA, 2015. Association for Computing Machinery.

[63] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 28(2):1–28, 2018.

[64] Mirko Viroli, Giorgio Audrito, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. A higher-order calculus of computational fields. *arXiv preprint arXiv:1610.08116*, 2016.

[65] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From distributed coordination to field calculus and aggregate computing. *Journal of Logical and Algebraic Methods in Programming*, 109:100486, 2019.

[66] Mirko Viroli and Ferruccio Damiani. A calculus of self-stabilising computational fields. In *Coordination Models and Languages: 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings 16*, pages 163–178. Springer, 2014.

[67] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, May 1992.

[68] Ruohan Yang, Lu Liu, and Gang Feng. An overview of recent advances in distributed coordination of multi-agent systems. *Unmanned Systems*, 10(03):307–325, 2022.

[69] Yaodong Yang. *Many-agent Reinforcement Learning*. PhD thesis, UCL (University College London), 2021.

[70] Ouarda Zedadra, Antonio Guerrieri, Nicolas Jouandeau, Giandomenico Spezzano, Hamid Seridi, and Giancarlo Fortino. Swarm intelligence and iot-based smart cities: a review. *The internet of things for smart urban ecosystems*, pages 177–200, 2019.

[71] Chen Zhang, Yu Xie, Hang Bai, Bin Yu, Weihong Li, and Yuan Gao. A survey on federated learning. *Knowledge-Based Systems*, 216:106775, 2021.

[72] Adrian Šošić, Wasiur R. KhudaBukhsh, Abdelhak M. Zoubir, and Heinz Koeppl. Inverse reinforcement learning in swarm systems, 2016.