University of Camerino

School of Advanced Studies

Doctor of Philosophy in
Computer Science and Mathematics - XXXVI Cycle

# A Framework for Simulating and Analysing Multi-Agent Systems in YODA


***Supervisor***

Prof. Michele Loreti

***PhD Candidate***

Nicola Del Giudice



Cycle XXXVI

*Do. Or do not...*

*There is no try.*

# ABSTRACT

Modern software solutions vary in dimension and domain scope and may be composed of a huge and heterogeneous amount of interacting entities. Each of these entities is characterised by a set of respective local or global goals. The entities have to operate and adapt according to a strategy, even cooperating or competing against each other. These interactions may produce many intricacies and unexpected results. Predicting these results should be supported by tools for simulating and analysing.

Operating entities are commonly referred to as agents and if we deal with multiple agents, this system is called Multi-Agent System. This is defined as "*a loosely coupled network of problem-solving entities (agents) that work together to find answers to problems that are beyond the individual capabilities or knowledge of each entity (agent)*". Agents may be enclosed in an environment and the dynamics between the actors may produce even more complexities, called "emergent behaviours". A proper definition of environment can help us define the whole system.

In this thesis, we introduce YODA, a novel framework for simulating agent-based systems. The framework allows the developer to define how the agents and the environment evolve by providing a set of formal rules and a specification language to model different case scenarios. YODA has been integrated into Sibilla, a Java-based tool for reasoning about Collective Systems, by taking advantage of the modular architecture of the tool. The thesis also includes a set of case studies to understand how YODA works with different scenarios. Additionally, we present two tools for further analysing Multi-Agent Systems. In the first case, we introduce Sequit a lightweight visualiser developed using the Unity Game Engine. On the other hand, we show an integration with the Model-Checking framework called GLoTL.

# LIST OF PUBLICATIONS

- **Del Giudice, N.**, Matteucci, L., Quadrini, M., Rehman, A., & Loreti, M. (2022, June). *Sibilla: A tool for reasoning about collective systems*. In International Conference on Coordination Languages and Models (pp. 92-98). Cham: Springer Nature Switzerland.
- **Del Giudice, N.**, & Loreti, M. (2022, September). *YODA: Yet anOther agent Description lAnguage*. In 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C) (pp. 82-87). IEEE.
- **Del Giudice, N.**, Matteucci, L., Quadrini, M., Rehman, A., & Loreti, M. (2024). *Sibilla: A tool for reasoning about collective systems*. Science of Computer Programming, 103095.
- **Del Giudice, N.**, Cruciani, F.M., & Loreti, M. (2024, June) *Visualisation of Collective Systems with Sequit and Sibilla*. In International Conference on Coordination Languages and Models (pp. 277-294). Cham: Springer Nature Switzerland.
- **Del Giudice, N.**, Loreti, M., Quadrini, M., & Rehman, A. (2024, October). *Monitoring local and global properties of collective adaptive systems*. In International Symposium on Leveraging Applications of Formal Methods (pp. 281-296). Cham: Springer Nature Switzerland.
- **Del Giudice, N.**, & Loreti, M. (2024) *Simulation and Analysis of YODA systems in Sibilla* (In Review).

# CONTENTS

**Abstract** iii

**List of Publications** v

**List of Figures** xi

**List of Listings** xii

**1 Introduction** 1
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 3
- 1.3 Structure of the Thesis . . . 5

**2 Literature** 7
- 2.1 MAS Models and Architectures . . . 9
- 2.2 MAS Tools and Libraries . . . 10
- 2.3 MAS Visualisers . . . 11
- 2.4 CAS Tools . . . 13

**3 Sibilla** 15
- 3.1 Sibilla Architecture . . . 16
- 3.2 Simulating with Sibilla . . . 19
- 3.3 Sibilla at Work . . . 21

**4 YODA** 25
- 4.1 YODA Computational Model . . . 26
  - 4.1.1 YODA Agents . . . 27
  - 4.1.2 YODA Environment . . . 28
  - 4.1.3 System Configuration . . . 29

4.1.4 YODA Semantics . . . 29
4.2 YODA Specification Language . . . 31
4.2.1 Parameters, Constants, and Functions . . . 32
4.2.2 Agents and Scene Elements . . . 33
4.2.3 Environment . . . 37
4.2.4 Configuration Declaration . . . 38
4.2.5 Measures . . . 40
4.2.6 Predicates . . . 41
4.3 Tracing and Simulating YODA with Sibilla . . . 41
4.3.1 Tracing the Running Scenario . . . 42
4.3.2 Simulating the Running Scenario . . . 43

**5 YODA at Work 47**
5.1 Flock . . . 47
5.1.1 Flock Specification . . . 48
5.1.2 Flock Simulation . . . 49
5.2 FinderBots . . . 52
5.2.1 FinderBots Specification . . . 53
5.2.2 FinderBots Simulation . . . 55
5.3 SEIR . . . 56
5.3.1 SEIR Specification . . . 56
5.3.2 SEIR Simulation . . . 58
5.4 Red-Blue . . . 60
5.4.1 Red-Blue Specification . . . 62
5.4.2 Red-Blue Simulation . . . 64
5.5 Concluding Remarks . . . 66

**6 Visualising YODA Simulations 67**
6.1 Implementing Sequit . . . 68
6.1.1 Architecture . . . 68
6.1.2 Features . . . 71
6.2 Visualising the Flock Scenario . . . 72
6.3 Visualising the Other Scenarios . . . 74
6.3.1 Visualising FinderBot . . . 75
6.3.2 Visualising SEIR . . . 76
6.3.3 Visualising Red-Blue . . . 76
6.4 Future Developments of Sequit . . . 78

**7 Monitoring YODA Models 79**
7.1 GLoTL Syntax and Semantics . . . 80
7.2 Monitoring GLoTL formulas . . . 84
7.3 Monitoring the Other Scenarios . . . 89
7.3.1 Monitoring FinderBot . . . 89

7.3.2 Monitoring SEIR . . . . . . . . . . . . . . . . . . . . . . . . 91
7.3.3 Monitoring Red-Blue . . . . . . . . . . . . . . . . . . . . . 93

**8 Conclusions 97**
8.1 Future Works . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98

**I Appendix Materials 101**

**A LPM Specifications 103**
A.1 Repressilator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
A.2 Predator-Prey . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104

**B YODA Specification Language 107**

**Bibliography 111**

**Acknowledgments 127**

# LIST OF FIGURES

1.1 Overview of the framework for simulating and analysing MASs in YODA . . . 3

2.1 A typical representation of an agent . . . 8

3.1 Sibilla Framework Architecture . . . 16
3.2 Quantities of mRNA and proteins over time in Repressilator . 22
3.3 Distribution of Sheep and Wolves after 100 time units (10 runs) 23

4.1 The basic Robotic Scenario . . . 26
4.2 Simulation Results of a Robotic Scenario . . . 43
4.3 FPT Analysis Results for the Robotic Scenario . . . 44
4.4 Reachability Analysis Results for the Robotic Scenario . . . 45

5.1 The three policies of a flock . . . 48
5.2 3 birds flock . . . 51
5.3 5 birds flock . . . 51
5.4 10 birds flock . . . 52
5.5 Finder Bots before (a) and after (b) finding the target zone . . 53
5.6 Simulation Results of FinderBot . . . 55
5.7 The SEIR model cycle . . . 56
5.8 Simulation Results of SEIR . . . 59
5.9 Simulation Results of SEIR with 7.5 `tolerance` . . . 60
5.10 100 Replicas of SEIR Model . . . 61
5.11 Red-Blue Scenario Interaction . . . 61
5.12 Simulation Results of Red-Blue . . . 65
5.13 Simulation Results of Red-Blue with 1000 agents . . . 65
5.14 100 Replicas of RB Model . . . 66

6.1 Sequit Architecture . . . 69

6.2 Scene Hierarchy of Sequit . . . . . 69
6.3 The two different available cameras . . . . . 71
6.4 UI of the visualisation speed . . . . . 72
6.5 The starting screen of Sequit . . . . . 73
6.6 The initial interface with the agent form . . . . . 73
6.7 The Flock at the beginning of the simulation . . . . . 74
6.8 The Flock after a while . . . . . 75
6.9 FinderBot in Sequit at the beginning . . . . . 75
6.10 FinderBot in Sequit after finding the target . . . . . 76
6.11 SEIR Model in Sequit . . . . . 77
6.12 Red-Blue Model in Sequit . . . . . 77

7.1 Syntax of GLoTL formulas. . . . . . 80
7.2 Derivable Logical Operators . . . . . 81
7.3 Horizon of Local and Global Formulas . . . . . 81
7.4 Global Formulas: Satisfaction relation . . . . . 82
7.5 Local Formulas: Satisfaction relation . . . . . 82
7.6 Function $\mathsf{after}_{\mathsf{L}}$ . . . . . 84
7.7 Operational Semantics of Global Formulas . . . . . 87
7.8 Acceptance and Rejection Criteria . . . . . 88
7.9 Global and Local Aggregation of Flock . . . . . 90
7.10 Global property applied to various cases of FinderBot . . . . . 91
7.11 Results of Monitoring Formula $\Phi_{gh}$ . . . . . 92
7.12 Results of Monitoring Formula $\Phi_{glh}$ . . . . . 92
7.13 Results of Monitoring Formula $\Phi_{gdi}$ . . . . . 93
7.14 Global property applied to various cases of Red-Blue . . . . . 94
7.15 Results of Monitoring Formula $\Phi_{gs}$ . . . . . 95
7.16 Results of Monitoring Formula $\Phi_{gls}$ . . . . . 96

# LISTINGS

3.1 Basic commands for simulating with Sibilla . . . . . . . . . . . 19
3.2 Repressilator model snippet in LPM . . . . . . . . . . . . . . . 22
3.3 Predator-Prey model snippet in LPM . . . . . . . . . . . . . . . 23
4.1 Parameters and constants variables in a Robotic Scenario . . . 32
4.2 Function in a Robotic Scenario . . . . . . . . . . . . . . . . . . 33
4.3 Attributes of a Robot Agent . . . . . . . . . . . . . . . . . . . . 34
4.4 Actions of a Robot Agent . . . . . . . . . . . . . . . . . . . . . 35
4.5 Behaviour of a Robot Agent . . . . . . . . . . . . . . . . . . . . 36
4.6 Obstacles in a Robotic Scenario . . . . . . . . . . . . . . . . . . 37
4.7 Environment in a Robotic Scenario . . . . . . . . . . . . . . . . 39
4.8 Configuration of a Robotic Scenario . . . . . . . . . . . . . . . . 40
4.9 Measure of a Robotic Scenario . . . . . . . . . . . . . . . . . . . 41
4.10 Predicate of a Robotic Scenario . . . . . . . . . . . . . . . . . . 41
4.11 Tracing Specification for a Robotic Scenario . . . . . . . . . . . 42
4.12 List of commands to start simulating the Robotic Scenario . . 42
5.1 Flock Scenario Specification . . . . . . . . . . . . . . . . . . . . 50
5.2 Flock Scenario Configuration . . . . . . . . . . . . . . . . . . . . 50
5.3 Tracing Specification for a Flock . . . . . . . . . . . . . . . . . . 51
5.4 FinderBot Scenario Specification Snippet . . . . . . . . . . . . 54
5.5 FinderBot Scenario Initial Configuration . . . . . . . . . . . . 55
5.6 SEIR Scenario Specification Snippet . . . . . . . . . . . . . . . 57
5.7 SEIR Scenario Configuration . . . . . . . . . . . . . . . . . . . . 59
5.8 Tracing Specification for SEIR . . . . . . . . . . . . . . . . . . . 59
5.9 Red-Blue Scenario Specification . . . . . . . . . . . . . . . . . . 63
5.10 Red-Blue Scenario Configuration . . . . . . . . . . . . . . . . . 64
5.11 Tracing Specification for Red-Blue . . . . . . . . . . . . . . . . . 64
A.1 Repressilator model in LPM . . . . . . . . . . . . . . . . . . . . 103
A.2 Predator-Prey model in LPM . . . . . . . . . . . . . . . . . . . . 104

# CHAPTER 1 INTRODUCTION

*Size matters not.*

*Look at me.*

*Judge me by my size, do you?*

## 1.1 Motivation

Modern computing solutions vary in dimension and domain scope. These may be referred to as *Cyber Physical Systems* (CPS) [1], used in medical contexts [2] or smart grids [3] where different physical components pass information about the enclosing environment, or as *Collective Adaptive Systems* (CAS) [4], like logistics or supplying networks [5]. A distinguished aspect of these systems is that they are often composed of a huge and heterogeneous number of interacting entities. Each of these entities pursues a set of *goals*, which can be *local* (related only to the single entity) or *global* (related to the collectivity). The entities have to operate and adapt according to a strategy and can cooperate or compete against each other in order to complete their tasks. Another important feature is that these entities do not operate under centralised control; thus, each one is responsible for its own actions. As we can imagine, by interacting and executing actions, these entities may produce many intricacies and even complex results: predicting such a quantity of outcomes can pose a difficult task to any user. Tools for simulating and analysing should support this activity in understanding said behaviours.

The presence of these entities is well-known in Artificial Intelligence and Complex Systems Theory [6]. However, these operating entities are commonly referred to as *agents*, or more precisely, intelligent agents [7]. Agents

differ from each other and are used in many research fields. When considering a group of agents executing some tasks, we consider a Multi-Agent System (MAS). The definition given by Durfee et al. in [8] describes a MAS as "*a loosely coupled network of problem-solving entities (agents) that work together to find answers to problems that are beyond the individual capabilities or knowledge of each entity (agent)*". The introduction of *agency* is what separates CAS from MAS: in the first case, we consider any possible interacting entity, while in the other case, the systems are strictly agent-based. Additionally, MASs pervade many real applications, from robotic scenarios, where behaviours are already scheduled, to more complex cases involving humans. Because of this, it is necessary to have methods to represent them as realistically as possible.

Over the years, researchers have devised different formalisms to accomplish this task, sometimes focusing on more specific domains while others trying to be more general purpose. Along with these languages, many tools have been developed to simulate agent-based systems. Each has its own architecture and supported languages, allowing the user to simulate and produce the intricacies of collective systems. A simulated scenario should take into account most of the aspects of an MAS, which is not always composed of mere agents.

Occasionally, the system considers the agents as entities enclosed in an environment. The latter's definition varied over time, as underlined by the authors of [9]. Some researchers treated the environment as the "external world", meaning that the environment is not an explicit part of the models or architectures [10]. Others saw the environment as a medium for coordinating agents, creating aggregate behaviour, as shown by the example described in [11] about aircraft landing. It is also useful to consider those systems based on stigmergic interactions [12], which are based on the concept of local information represented as marks (some associate them even to ants' pheromones) that can be used to retrieve information about the surroundings.

Being able to specify MASs is still a relevant problem, especially when combining environmental dynamics. Interactions among these components may produce complexities or even unexpected results, called *emergent behaviours* [13]. This activity should be supported by tools (such as languages and software) that provide comprehensive knowledge of the systems. In some cases, it could be possible that the agents are "blind", meaning they can not have direct access to external information [14]. The reasons could range from fallible sensors to high component costs. In addition, they cannot correctly update themselves or communicate useful data with others. Thus we need to depute an appropriate entity to overcome this necessity. As we will see in this thesis, the enclosing environment—which should not be considered merely the world around it but also an interacting entity—can resolve this

issue.

## 1.2 Contributions

This thesis presents a novel framework dealing with the development and simulation of agent-based systems. The framework is conceived to allow anyone to start designing MASs using the provided language, load the specification into the simulator, and perform the necessary graphical and formal analyses.

Figure 1.1 shows an overview of the framework proposed in this thesis and how each component interacts with the others. In the scheme, continuous lines represent the components of the thesis's main contribution. On the contrary, dashed lines depict components that were already developed and that have been integrated into the proposed framework.

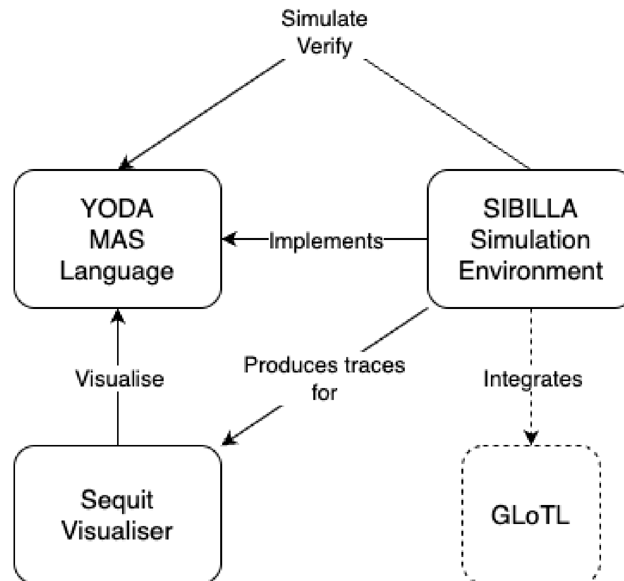



Figure 1.1: Overview of the framework for simulating and analysing MASs in YODA

The thesis' contributions can be summed up in the following manner:

- First, we introduce the Sibilla simulation tool [15, 16], a framework developed in Java to simulate Collective Systems. The tool has been conceived as modular software acting as a container supporting different specification formalisms. Additionally, it allows the user to integrate multiple analysis techniques, such as *optimisation* [17] or *model-checking* [18]. The key feature of Sibilla is customizability. Indeed, Sibilla is already equipped with a set of languages to design collective systems and a set of functionalities to perform different quantitative

analyses. To explain the available features, we use two examples, one based on a *Repressilator* [19], an auto-regulating genetic network. In contrast, the other example takes inspiration from the Lotka-Volterra *Predator-Prey* model [20], where a pack of wolves hunts down a herd.

- Another contribution is the development of YODA (*Yet anOther agent Description lAnguage* multi-agent language [21, 22]. YODA is a language conceived to specify MASs and is equipped with both a computational model and a specification grammar. The language comes from a long tradition of research on *Formal Methods* [23, 24], which are defined as "*mathematical techniques, often supported by tools, for developing software and hardware systems*". Formal Methods are used to rigorously specify software, stating what it should precisely do. On the other hand, they should also avoid constraints on how software should achieve such results. Indeed, YODA's computational model shows what agents and environment should do to proceed into the next state. To help the reader understand the language, we use a running example based on a *Robotic Scenario* [25]. This considers a group of robots that, given a starting point, must reach an area at the opposite end of a grid map. The robots can decide autonomously in which direction to avoid obstacles and complete their task. We report the results of simulating the robotic model using Sibilla.

- Together with this example, we specified more complex cases using YODA, such as the *Flock behaviour*. This model was inspired by the well-known *Boids* scenario designed by Reynolds in 1987 [26]. Agents (the birds or any entity displaying swarm behaviours) proceed in a certain direction while adopting one of three possible policies:

  **Alignment** Tending to maintain the same direction as the neighbours;

  **Cohesion** Keeping a certain distance from the others;

  **Separation** Avoiding collisions with each other.

  Another complex scenario used to display YODA capabilities is the *FinderBot*, which recalls those where agents need to retrieve an injured civilian [27] or eradicate an invasive grass [28]. The proposed model is useful to understand how the behaviour can drastically change in YODA-based models. Normally, agents roam, in a random manner, the environment looking for a target. Once they reach the interested area, they stop moving and signal the target's position to the other agents. The third example is based on the *SEIR* [29], a model used to understand epidemic scenarios. In this example, a group of susceptible individuals may get in contact with infected ones. In this case, they

become exposed and then infected themselves. After a certain time, the illness ends and the individual recovers. Finally, the last example is called *Red-Blue* and is based on consensus scenarios [30]. Here, we imagine two types of supporters (red and blue) who can change parties when there are more of the same side around them.

- As we said, developing complex systems may produce unexpected behaviours. These may be involuntary from the developer's perspective, and they can be investigated only using quantitative analysis tools. The final piece of the thesis regards integrating tools to better comprehend the simulation results. Relying only on static visualisation, such as plotted graphs, could not be enough. For these reasons, we created Sequit [31], a lightweight visualiser developed with *Unity*. The idea is to have a tool that can be used immediately, even by non-experts, given a *.csv* file with simulated agent traces. The visualiser permits us to appreciate the evolution of the system even further.

- Verifying properties play a significant role in the domain of formal methods. This technique is fundamental when we need to ensure various characteristics in complex systems, such as robustness, consistency, or reliability. To perform such analyses in YODA, we took advantage of the GLoTL framework [32]. This logical framework was already integrated into Sibilla, and we used it to verify some properties of one of the proposed examples. We will discuss the semantics and syntax of GLoTL and how we implemented it to model the four additional case scenarios.

## 1.3 Structure of the Thesis

The remainder of the thesis is divided as follows.

- Chapter 2 contains an overview of the related works we took as inspiration for the development of the framework. We introduce the Multi-Agent Systems topic in Section 2.1 by looking at some important languages. These gradually deal with the concept of environment as a main component of an agent-based system. Then, Section 2.2 presents some tools focused on MASs we found interesting to understand the connection between YODA and Sibilla. In Section 2.3, we consider more tools that allow the user to visualise the results of simulated agents as a confrontation with Sequit. Finally, we discuss general CAS tools in Section 2.4, posing the foundations of the Sibilla simulator.

- Chapter 3 introduces the Sibilla simulator, a Java-based modular tool we used to simulate the agents. In Section 3.1, we start by explaining

the tool's architecture and how the components interact. Section 3.3 follows with a brief discussion of two examples: one about a biochemical mechanism, the other on predator-prey behaviour. We conclude the chapter with Section 3.2, which explains the commands for simulating and tracing with Sibilla.

- YODA is actually presented in Chapter 4. In Section 4.1, we introduce the computational model of YODA. This is used to understand how a system developed using YODA works and evolves. Section 4.2 contains the specification rules for designing MASs with YODA, while Section 4.3 shows how we can simulate YODA models using Sibilla functionalities.

- Chapter 5 contains a set of case studies based on common scenarios to show how the language works. The first case scenario, presented in Section 5.1, is based on the flocking behaviour, where the agents move around in a coordinated way. The FinderBots example, shown in Section 5.2, describes a scenario where a group of agents has to find a target. Section 5.3 presents the SEIR scenario, which is used to mimic epidemics and illness transmission. Finally, the last example, based on a consensus scenario, is discussed in Section 5.4.

- In Chapter 6, we introduce Sequit, a *Unity*-based visualiser to replay collective system via agent traces. We show in Section 6.1 its architecture and how the main of the tool interact with each other. The discussion is followed by the visual representation of the four main scenarios.

- Chapter 7 presents some analyses using model-checking algorithms. Specifically, we combine the proposed YODA-based scenarios with a logical framework for model-checking called GLoTL. This framework permits reasoning about global (environmental) and local (single behaviours) properties of collective systems. Here, we show the plots related to the satisfaction probability of formulas of the four scenarios.

- We discuss the conclusion by summing up the presented work in Chapter 8. Here, we also discuss the future directions of the YODA framework.

# CHAPTER 2

# LITERATURE

*In a dark place*

*we find ourselves*

*and a little more knowledge*

*lights our way.*

Over the years, agents have received many definitions and characterisations [33]. Even if the term *agent* is widely used, there is no universally accepted definition. Commonly, an agent is referred to as something that resembles what is depicted in Figure 2.1. It interacts with an environment by receiving observations, which the sensors will detect. Then, depending on the agent's behavioural rules, these data are used to activate actuators, allowing the agent to perform an action that will affect the environment. Nevertheless, we can find many valid definitions in the literature.

One of the most quoted definitions is given by Woolridge and Jennings [7], who identify two different interpretations for agents. The first one, called "weak", is uncontentious and easy to understand. In this case, an agent can be an hardware or, more commonly, a software entity that is autonomous, social, reactive, and proactive. This classification is appreciated by many researchers and is used in disciplines like *agent-based software* [34], or different kinds like software robots (also called *softbot*) [35]. The "strong" definition of agent, which is more specific than the previous one, is more suitable for those who work in AI fields. In addition to the aforementioned characteristics, researchers consider the agent to be a more conceptualised system or even implementation using concepts applied to humans. Some agents use mentalistic notions, taking advantage of knowledge or desires (some even

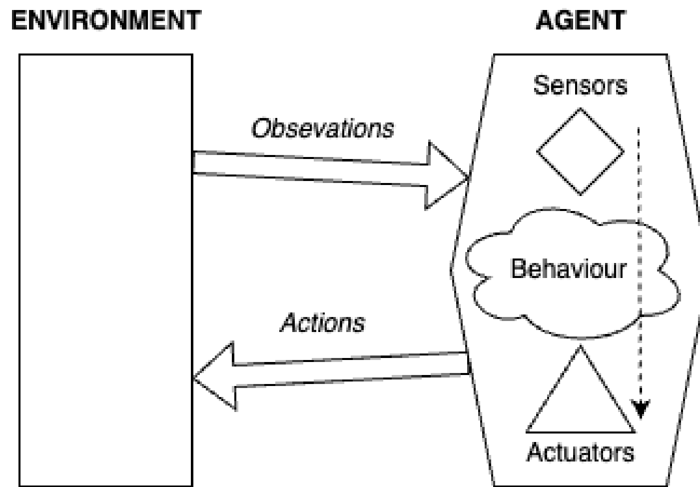



Figure 2.1: A typical representation of an agent

call these emotional), while others are visually represented with icons and human-like attributes when human-computer interfaces are discussed.

Russell and Norvig give another well-recognised definition in [36], where a *rational agent* is an entity that "*for each possible percept sequence, an ideally rational agent should do whatever action is expected to maximize its performance measure, based on the evidence provided by the percept sequence and whatever built-in knowledge the agent has.*" By the authors' admission, this definition could lead to performing irrational actions. The rationality of an agent should consider a set of actions for obtaining useful information.

That said, agents usually do not act alone in the environment and must consider other working entities. When this happens, we are talking about Multi-Agent Systems. We recall that the authors of [8] consider a MAS as a "*a loosely coupled network of problem-solving entities (agents)*". These may "*work together to find answers to problems*" or even compete, reacting to others or their environment according to behavioural rules, ranging from basic primitives to more complex AI operations. MASs are used, for example, in manufacturing scenarios, to model processes and automation [37], in power engineering, to understand the management of electricity [38], or even in hospital scheduling, to arrange patient appointments [39]. Some MASs are also used to aid decision-making [40] due to the increasing complexity in systems or other issues, like retrieving data in real-time.

As this thesis concerns the development of a framework to simulate and analyse MASs, we would like to introduce some related works and similar solutions in this chapter. We focus on MASs languages in Section 2.1, where agent-based architectures are shown to understand the development process of YODA. Section 2.2 introduces highly specialised tools to simulate and anal-

yse MASs. MAS visualisers and 3D renderers are discussed in Section 2.3 to acknowledge similar instruments of Sequit. Before moving into the next chapter, which deals with our main tool Sibilla, we would like to show some general-purpose simulators relative to Collective Systems in Section 2.4.

## 2.1 MAS Models and Architectures

The development of a MAS language should consider the characteristics and unique traits of existing languages and formalisms. Depending on the type of specifications and requirements, some of them may be more suitable than others. If we need a more context-agnostic language, we can consider more general ones. Otherwise, we must rely on those more focused on specific domains. For our means, this section considers only a restricted group of MAS languages that inspired the formalisation of YODA.

We start talking about the *BDI model*, discussed in [41] and among the most known MAS architectures. The acronym BDI stands for *Belief, Desire, and Intention*, intended as three different attitudes that an agent can have during its execution, and it is inspired by Bratman's model [42]. In this architecture, an agent's state is represented by *beliefs*, which can refer to the agent itself or other entities. This term, in contrast to *knowledge*, accounts also for incorrect information that agents may have. To represent what the agent wants to accomplish, *desires* are used. These mental attitudes are desirable and not mandatory: if we need to define something mandatory and not concurrent with other desires, we must include a *goal*. Finally, the *intentions* represent what the agent has chosen to perform to proceed into the next state. When a sequence of action is activated, the agent executes a *plan*. This model, however, reserves some issues, ranging from BDI multimodal logics practical utility, discussed in [43], to the ability of agents to adapt and learn from the past [44].

The definition of *Interpreted Systems* (IS) deeply inspired the development of YODA and has been explored initially in [45] and further discussed in [46]. They are used for *Situation Analysis*, which can be defined as contextual analysis executed on processes to achieve some kind of knowledge for a collection of agents. These systems comprise a group of agents $\mathcal{A}$, plus the special one $e$ denoting the *environment*. Each agent's local state is denoted by $l_i$ (while $l_e$ for the environment), and all local states belong to the *global state* $s$. In an IS, actions allow the system to change and are denoted by $ACT_i$, the set of all possible actions an agent can perform. The evolution of the whole system is based on the results of each agent's local steps, including the actions performed by the environment.

It could be interesting to the work presented in [47], where the authors propose a generic model that can be used to describe *Situated MASs*. *Situat-*

*edness* is a property expressing that an agent is not isolated but exists in an environment. However, situated MASs consider the agents as social entities, and situatedness expresses the local relationships between them and the objects in the environment. These relations drive the evolution of the system. Further studies on the argument [48] underline how some aspects of agent-based systems "*that conceptually do not belong to the agents*" should not be "*assigned to, or hosted inside agents*". This latter paper lays the foundation for the idea that these characteristics, like topologies, specific resources, or indirect coordination, need to be explicit.

Other MASs rarely take an explicit environment into account. As a derivation from the definition of Situated MAS, an interesting point is made in [9], where the authors claim that the environment should be considered a *first-class abstraction*, thus being explicit. This environment permits agents to exist, act, and behave according to their rules, and it is also used to mediate between entities and available resources. For the authors, another characteristic of a first-class environment is that it should be exploitable by designers and developers. This means that engineers should think of the environment as a building block for their MAS. Moreover, the environment should have separate responsibilities, like maintaining dynamics unrelated to agents, being accessible by the agents, or defining rules.

## 2.2 MAS Tools and Libraries

Simulation and analysis of MAS are done through the use of specific tools. These aim to provide deep insight into the intricacies of agent-based models, producing traces and results from simulations. In this section, we survey some MAS-related tools and libraries.

Among the most famous tools for agent-based system simulation, we find *JADE* [49, 50], a FIPA-compliant[1] tool aiming to facilitate the development of agent applications. The tool provides programmers with a set of functionalities, such as management systems, load distribution with other clients, programming interfaces, and more. JADE also tries to be efficient in terms of resource consumption and keeps runtime overheads low, paying runtime costs only when a feature is used.

Regarding efficiency, a more recent solution is *Repast Symphony* [51, 52], where common tasks are automated and boilerplate code is avoided. This tool is based on Repast3, a group of open-source libraries for agent modelling, and keeps all the features of the old version. One of Repast Symphony's design goals is to separate (and maintain) model specification, execution, data storage, and visualisation. For this reason, the authors rely on Eclipse

[1] http://www.fipa.org/index.html

to develop their framework. Repast Symphony also provides graphical user interfaces for agent behaviour modelling.

With a long-term support, *MASON* [53] is one of the oldest agent-based libraries. It has been developed as a general-purpose library for Java and consists of two parts. On the one hand, we have the simulator, while on the other, we have all the tools to visualise data via a graphical interface. This separation allows the user to run efficiently on back-ends and view and modify checkpointed simulations. The authors are working to improve the tool, by extending the available features and optimising analysis tools [54].

*SPADE 3* [55] aims to include various technologies, acting as a middleware for Multi-Agent Systems. The tool aims to find a solution to a set of issues: 1) relying not only on closed or local communication, 2) scalability for massive MASs, 3) allowing human interactions, and 4) interoperability with IoT systems. It focuses on XMPP, an XML-based protocol, that allows message exchange between entities and provides a notification system.

The *JaCaMo* framework [56, 57] puts together three different agent-based languages (Jason, CArtAgO, and Moise) to let programmers simplify the development of more complex systems. With JaCaMo, authors go beyond simple integration and provide a support application for synergistically developing agents, environments, and organisations.

The solutions above are extremely specialised in certain MASs architectures and are not suitable for our objectives: we need a tool that does not rely on a single language or architecture. For our purposes, we worked with Sibilla, a tool we used as a simulation environment and that has been discussed initially in [15] and in detail in [16]. Sibilla is a modular tool that permits adding different or new languages, by developing the necessary classes according to its architecture.

## 2.3 MAS Visualisers

Among the techniques for analysing MAS, we can include the possibility of visualising simulation data. Many approaches combining simulation and 3D visualisation already exist in the literature. Similar to the tools seen in the previous section, those used to visualise agent-based systems may vary in architecture, and each one specialises in its own way of representing the agents. We want to introduce some existing solutions that may be useful to understand the general overview of the topic in this section.

The first platform we would like to cite is *AnyLogic* [58, 59], a proprietary software used in multiple domains (from healthcare to risk events or even transportation) for simulation. AnyLogic is a programming and simulation environment based on Java language. The tool focuses mainly on agent-based and business simulations, but it can also be used in other con-

texts. In addition, many existing libraries are natively available, like those for simulating pedestrians or railing systems.

Likewise, *StarLogo TNG (The Next Generation)* [60], and the newer, web-based version *StarLogo Nova*, is a proprietary tool (in this case owned by MIT). It relies on visual scripts to ease the development of agent-based systems. As the name may suggest, the tool is derived from *StarLogo* [61], an agent-based language, which in turn is inspired by *LOGO*. However, StarLogo TNG has been discontinued, and the last update for the Nova version dates back to 2021 (at the time of writing).

Moving to open source solutions, *GAMA* [62, 63], an "agent-oriented" tool based on Eclipse, is an interesting choice. The platform relies on a specialised agent language, called *GAML* (GAma Modeling Language), which allows even non-expert users to declare agent species and give them a behaviour. GAMA provides a user interface to write and run agent models and 3D displays to render simulations. Moreover, the simulator is interactive, meaning the user can inspect the entities, interact with the display, and more.

*Gazebo* [64, 65] is also an open-source software, even if it is more oriented to Multi-Robot Systems and robotic simulations. It integrates *ROS* (Robotic Operating System) to model systems and uses *DART* as a physics engine. Gazebo allows the user to interact with the tool via different interfaces (command line, web, and graphical) and can be extended with other tools or even graphical plugins.

A recent solution is the one given by the authors of [66]. In this paper, a graphical user interface is presented to support *FCPP* [67] (Field Calculus implemented in C++), an efficient library for working with aggregate programming. The authors state the necessity of a more readable solution and not limited to numeric statistical information. For these reasons, the proposed tool aims to provide an interactive real-time visualiser for representing aggregate systems in 3D. The tool also aims to ease the user experience and allow control and interaction with the simulation.

The usage of common Game Engines (GE), such as *Unity* or *Unreal Engine*, for simulating MASs has also been questioned in [68]. Here, the authors propose a roadmap for integrating GEs and MASs and exploiting the features of the first ones. These should be used for the rendering properties and the eventual features that may derive from this union. We can find some plugins that can be used in pairs with Unity3D to perform crowd simulation. For example, the authors of [69] take advantage of BDI architectures, while in [70] a plugin to use the *Menge* framework with the GE is presented.

## 2.4 CAS Tools

Collective Adaptive Systems can be considered a generalisation of MAS. The main difference between the two kinds of systems is that, unlike in MAS, the interacting components in CAS may not be agents. This section presents the main tools proposed to model and forecast the behaviour of CAS. Each of these tools focuses on a specific formalism used to model CAS using distinguishing paradigms and forms of interaction. We will see in the next Section that the tool used in this thesis, named Sibilla, has been conceived as being language agnostic. This means that any new language can be integrated.

*ALCHEMIST* [71] is a software which mainly targets scenarios where agents interact in terms of biochemical-inspired primitives. The tool is based on an optimised version of the stochastic simulation algorithm designed by Gillespie, and it is extended with the possibility to deal with the changing environment (updating rules and items). The tool maintains a pretty generic simulation framework, allowing other developers to use its APIs in other applications, like pervasive computing or even swarm scenarios. Indeed, in [72], the authors use ALCHEMIST's APIs to simulate a complex swarm behaviour regarding reusable functional blocks.

Concerning the analysis of logistics and transportation systems, we can include in the discussion *RinSim* [73], a flexible and modular open-source simulator addressing this type of application. Parts of the simulation defining the addressed problem are called "models", which can be combined due to their independence. Of course, these models are also characterised by constraints to avoid conflicts.

Analyses of quantitative systems are also performed by frameworks like *CARMA* [74]. This is a process algebra with a rich set of communication primitives supporting both unicast and broadcast communications. The richness is conceived to express the spatial characterisation of CASs. The language supports scalable analysis techniques with its semantics, which are either discrete or continuous. Supporting tools have been developed to ease the use of CARMA, such as a plugin for Eclipse [75], and the language has also been used to mesoscopically model and simulate pedestrian movements [76].

While working on formal models, it is useful to consider model-checking techniques to validate said systems. Among the most known available model-checkers, we can find *PRISM* [77], a probabilistic model-checking tool. Different from classical model-checkers, this is based on probabilistic models, meaning that it considers the probability of making a transition rather than its existence. The PRISM modelling language is also supported by *STORM* [78] probabilistic model-checker. This recent tool succeeded in resolving more instances and faster than the competitors. Moreover, it has been conceived to be modular, permitting the adaptation of algorithms or other features.

# CHAPTER 3

# SIBILLA

A Collective Adaptive System [4] is a large group of interactive heterogeneous entities or components. These cooperate and compete depending on their properties to reach local and global goals. Predicting the global behaviour of such systems may be complicated, and interactions among their components may introduce new, and sometimes unexpected, *emerging behaviour*. Thus, robust modelling techniques should be used to describe CASs and reason about their behaviour in qualitative and quantitative terms. In the previous chapter, we saw different tools and languages highly specialised in MASs. More generally, different approaches have been introduced and successfully applied to support the design, analysis, and development of CAS. These approaches, mainly based on formal techniques and tools, are often based on *high-level specification languages*. These languages rely on *Process Algebras* [79] and their *stochastic variants* [80, 81] and are derived from a long tradition of *process specification languages* [82]. The latter has been used to specify a multitude of *Domain Specification Languages* like those related to *Global Computing* [83, 84], *Service Oriented Architectures* [85, 86], or even *Biological Systems* [87, 88] That said, a formalism should provide appropriate abstractions and linguistic primitives to deal with CAS's specific features. For example, SCEL [89] and AbC [90] can be used to model interacting *attributes-based* agents and their behaviour.

To overcome the limits of language-based tools, Sibilla[1] [15, 16] has been developed. The tool provides a standard set of features that can support different specification formalisms and integrate multiple analysis techniques. One of the main features of Sibilla is that it is not developed to support only one specification language but can be integrated with new languages.

[1]The tool is available on GitHub at `https://github.com/quasylab/sibilla`.

Currently, the tool is equipped with APIs for

- *system simulation* to forecast systems' behaviour;
- *transient analysis via statistical model checking* to prove the correctness of models;
- and *modelling different CASs* to define various types of systems.

This chapter describes Sibilla inner workings by showing its modular architecture and three main components. Then, we proceed to explain how the tool can be used to simulate CASs using two examples: one based on a repressilator machine, and the other on a predator-prey scenario. Both are specified using the Language of Population Models.

## 3.1 Sibilla Architecture

As illustrated in Figure 3.1, Sibilla is based on a modular architecture consisting of three layers. The first one is the *back-end*, which comprises a set of interfaces and classes used to implement the basic blocks on top of which Sibilla features are built. Then we have the *front-end* classes, providing the environments used to interact with Sibilla. Finally, the interactions between the other two layers are coordinated by *runtime* layer. Sibilla has been implemented using Java and is based on Gradle: this allows the user to build and deploy the system on multiple operating systems.

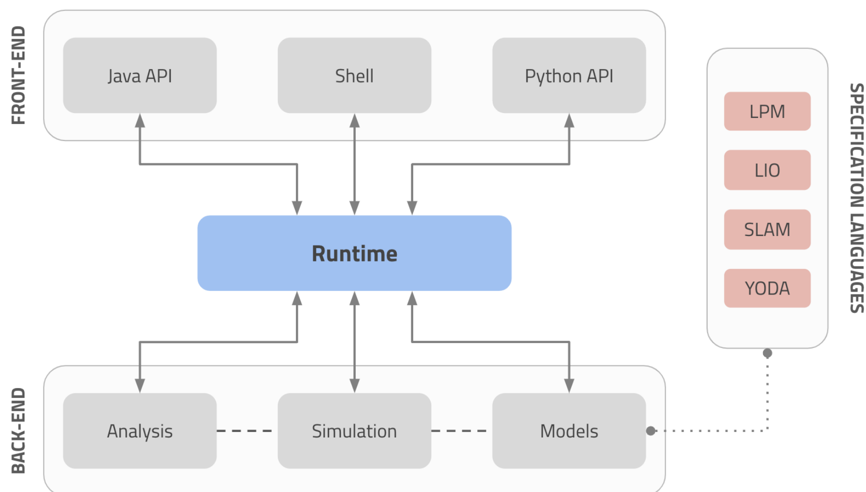



Figure 3.1: Sibilla Framework Architecture

### Back-End

The back-end of Sibilla is composed of three components: *Models*, *Simulation*, and *Analysis*. The first component is responsible for providing interfaces and classes, which will be used to describe a *Stochastic Process* [91], which is formally defined as a collection of *random variables* $\{X(t) | t \in T \subseteq \mathbb{R}_{\geqslant 0}\}$, representing values on *measurable set* $S$, the *state space* of the stochastic process. In the formula, the random variable $X(t)$ describes the state of the process at time unit $t \in T$. Depending on the $T$, which can be *discrete* or *continuous*, or the properties of the considered process, we can have different types of stochastic processes. In addition, the component *Models* provides utility classes for handling specific features of the considered models, e.g. *transient analysis* of *Markov Chains* [92]. These classes provide the base on which specification languages can be integrated into Sibilla.

Defining an exact numerical analysis may be difficult or even impossible when considering a system composed of many entities. This issue is mainly due to the problem of *state space explosion* [93]. To avoid this issue, a set of classes is provided to support the simulation of stochastic processes. These permit the sampling of a *path* from the model, representing a possible computation/behaviour of the model. Moreover, the same classes provide methods to extract *measures* from a *path* and to collect statistical information. Samplings and statistical analyses rely on the widely used and well-known API *The Apache Commons Mathematics*[2]. Unfortunately, simulations may require a remarkable computational effort, and efficient use of computational resources is crucial. Because of this, Sibilla simulation classes follow a *multi-threading* approach based on *Java Concurrency API*. Different *scheduling policies* are available to execute *simulation tasks*, and can be tailored to fit with the *parallelism* of the hosting architecture [3].

The final component of the architecture is the module responsible for performing analysis techniques used to extract data. These are collected from the Sibilla simulator to asses the quantitative properties of analysed systems. These analyses rely on *statistical inference techniques*. Currently, the integrated techniques are the computation of *first-passage-time* [94], *reachability analysis* [95], and *monitoring of properties*. The *first-passage-time* is used to estimate the average amount of time needed by a model to reach a given *condition*, which is stated in terms of a *predicate* on the states of the considered process. *Reachability analysis* permits estimating the probability that a given set of states (identified by a *condition*) can be reached within a given amount of time by passing through states satisfying a given predicate. Finally, *monitoring of properties* is used to compute the probability that a

[2] https://commons.apache.org/proper/commons-math/

[3] The impact of *scheduling policies* is reported at https://github.com/quasylab/sibilla/wiki/Multi-threading-simulation

given property, specified via a *Signal Temporal Logic* formula, will be verified at a given time.

### Runtime

The *Runtime* layer coordinates and controls those interactions between the front-end and back-end. If we compare it to the standard *Model-View-Controller* (MVC) pattern [96], this layer is similar to the *controller* component, and it is used by all the components in the Sibilla *front-end*. From the composition point of view, the Runtime is structured in *modules*, each of which manages a *specification languages*. A module is identified by a unique *name*. This is used to *select* the specification language while the user is working in the runtime environment. Each specific module extends the abstract class `AbstractSibillaModule`, which implements some methods of the `SibillaModule` interface class. The `AbstractSibillaModule` class provides the operations that are common to each specification language, such as retrieving information from the specification (parameters, constants, initial configuration names), controlling the simulation parameters, or executing different analyses like the reachability or the *First-Passage-Time*. This modular architecture simplifies the integration of new specification languages into Sibilla.

### Front-End

The *front-end* layer provides the interfaces that allow users to interact with Sibilla. The software and the user interactions can occur via the command line tool, denominated Sibilla *shell*, or at a code level, using Java or Python scripts. The Sibilla shell provides a simple script language that can be used to interact with the tool. It can be used interactively or in a *batch mode* to execute saved scripts.

The *Python front end* permits using all the provided features in a Python script. This component relies on the *Pyjnius* library [97] that simplifies the interaction between Python and Java classes. This front-end allows to use many of the available Python libraries, like Matplotlib[4] that simplifies data visualisation. Moreover, it also allows the use Sibilla within a web-based IDE such as Jupyter notebook/lab [98] and Google Colaboratory [99].

Finally, Sibilla can be integrated into other Java applications. After having imported the module *Runtime*, Sibilla features are available employing class `SibillaRuntime.java`. This class permits accessing all functionalities provided by the Sibilla the back-end, together with the methods for specifying simulation and analysis tasks.

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

## 3.2 Simulating with Sibilla

Sibilla provides a set of commands that can be used to interact with the tool. These permit selecting the *module* to use, loading a specification from a file, setting the correct parameters, and performing analyses. The main commands that can be used to perform a simulation in Sibilla are reported in Listing 3.1.

```
module "<name>"
load "<filename>"
init "<confname>"
add all measures
deadline <real>
dt <real>
replica <int>
simulate
save output "<foldername>" prefix "<prefix>" postfix "<postfix>"
```

Listing 3.1: Basic commands for simulating with Sibilla

Command module "<name>" is used to select the module to use and we are telling to Sibilla which language we want to use. For example, if we want to use the YODA module, we can type in the CLI the command module "yoda". After that, the command load "<filename>" can be used to load a specification file from the given path, e.g. if we want to use a given model developed in YODA, we can use load "myModel.yoda". Then, we must select the initialisation configuration from which the simulation will start. This operation is performed by using the command init "<confname>", where "<confname>" is the model configuration name we want to select. Obviously, the configuration should be included in the model and the name should be unique.

To configure the simulation, we need to set some parameters. The command add all measures permits us to add every specified measure. These are functions used to collect data from the simulated model. If we want to capture only one measure, we can use add measure "<name>": in the available languages of Sibilla, we can declare the category measure followed by an identifier, which can group a set of chosen entities. To set the deadline of the simulation, command deadline <real> is used. This permits determining how long the simulation will last. Finally, dt <real> is used to select the sampling time while replica <int> sets the number of replicas to perform the simulation.

The command simulate can be used to start the simulation while the command save output "<foldername>" prefix "<prefix>" postfix "<postfix>" permits saving data to an external file. With this last command, we choose the name of the output folder, the prefix, and the postfix to differentiate the saved files.

The Sibilla shell interpreter has a limited amount of commands. The Python-based front-ends represent more complete solutions. These are also considered commands to work with plotter and other Python tools. An interested reader can view the complete command list on the GitHub wiki page.

Simulation can be used to sample a given number of computations from a given model and collect statistical data.

Sometimes, it is helpful to render the result of a single run of a model. For this project, we decided to add to Sibilla a way to produce single traces of the agents. For this reason, the *trace* command has been implemented to capture the agent state at each instant of the simulator runtime. The trace specification requires little effort from the developer, and it helps the Sibilla .csv writer to understand how data should be printed. The following syntax shows how a trace specification can be expressed:

<*name*> **=** <*expr*> **;**
<*name*> **=** {<*whenBlock*>} **;**

A trace specification code line consists of two main parts: the label and an assigned value (which can be an element of the expression syntactic category or a multiple cases block). The labels identify what should be captured at each time instant and can not be changed. Apart from the **Time**, the assignable labels are the following:

- **x, y, z**, the three axes that are used to Vector3 variable used to move the agent in the space;
- **direction**, a variable used to rotate the agent in a certain direction;
- **shape**, used to change the shape (sprite or model);
- **colour**, a variable used to change the colour of the agent.

The assigned value can refer directly to a variable of a specified model or other values. Additionally, the assignment is done automatically if the variable has the same name as the label. However, if we want to switch between certain values, we need to use the syntactic category <*whenBlock*>, which is declared in the following manner:

(**when** <*guard*> **:** <*expr*>**;**)*
**otherwise** <*expr*>**;**

Depending on the guard, which usually refers to a model variable, a specific value is given to the label (otherwise, the default case is activated) Indeed, this block is often used to change the shape and colour of the agents.

A trace specification is called at runtime using the corresponding command. This command should be declared in the following manner:

'trace' (input=STRING) 'in' (output=STRING) ('h' '=' header=expr)?;

As we can see, we can identify three different parts. First, we decide which trace specification we would like to use for producing the .csv files. Then we indicate the output folder. Finally, we need to decide if we want the header row or not. At the end of the command, we can indicate this by typing `h = true/false`.

## 3.3 Sibilla at Work

We have already remarked that the simulator does not focus on a specific formalism to model systems' behaviour but is conceived to integrate multiple specification languages. At the moment, three languages are supported other than YODA: *Language of Interactive Objects* (LIO), *Simple Language for Agent Modelling* (SLAM), and *Language of Population Models* (LPM). Focusing on the latter, it consists of agents belonging to a given set of *species* [100]. A system evolves using *reaction rules* describing how the number of elements of the different species changes. Rules are applied with a *rate*, a positive *real value* that depends on the number of agents in the different species. The stochastic process associated with an LPM specification is a *Continuous Time Markov Chain* [101]. Population Models have been widely used to model different systems in different application domains ranging from ecology [102] to epidemics [103].

We consider two examples, based on the LPM, to show how Sibilla can execute simulations to support analysis of complex scenarios. In the first case study, we will consider the *Repressilator* model, a genetic regulatory network that consists of a loop of interactions between at least three genes. Then, we will discuss a *Predator-Prey* scenario, where we represent a flock escaping from a pack of wolves and moving inside a grid. These two simple and well-known scenarios permit appreciating how Sibilla can be used to simulate and analyse data. The full specifications of the proposed models can be found in Appendix A.

#### Repressilator

A Repressilator is a genetic network auto-regulated via a feedback loop with at least three genes in a "rock-paper-scissors" fashion. The model, initially described in [19], can be specified in terms of the rules of a population model. In our specification (an excerpt is shown in Listing 3.2), we declare the six species: three mRNA segments, called $X$, $Y$ and $Z$, and their three corresponding proteins, $PX$, $PY$ and $PZ$. Moreover, *VOID* is an artificial species used to simplify the model and is involved when we need rules about *degradation* (segments and proteins degrade), *translations*(production of proteins),

and *transcription* (production of mRNA segments). Figure 3.2 shows the results of 25 simulation runs of 250 time units. It has quantities of mRNA segments and the protein produced over time.

```
param ...

const startY = 20;
const startVOID = 1;

species VOID;
species PX; ... species X; ...

rule degradation_of_X_transcripts {...}
...
rule translation_of_X {...}
...
rule degradation_of_X {...}
...
rule transcription_of_X {...}
...
system initial = Y < startY >|VOID < startVOID >;
```

Listing 3.2: Repressilator model snippet in LPM

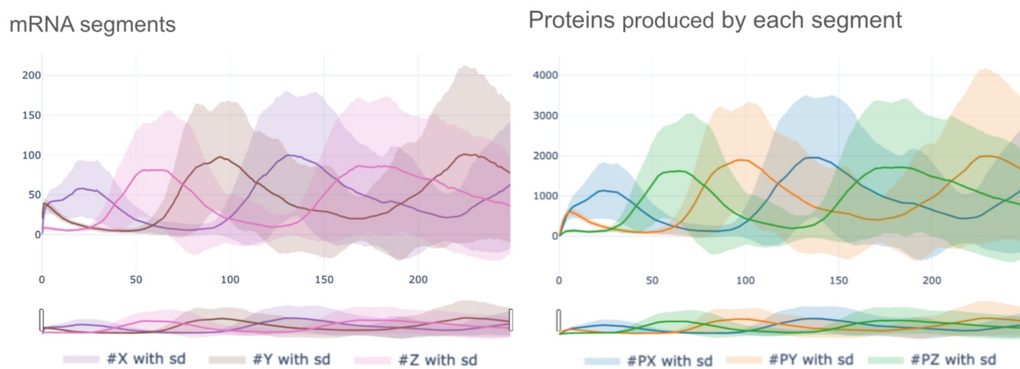



Figure 3.2: Quantities of mRNA and proteins over time in Repressilator

### Predator-Prey

This classical model describes the behaviour of two kinds of animals, one acting as predators (wolves, W) and the other as prey (sheep, S). In the specified model, of which we find an excerpt in Listing 3.3, we assume that only prey die (eaten by the wolves), differently from similar models like the Lotka-Volterra [20]. We focus on the behaviour of Predator-Prey. We also assume that all animals move on a grid of dimensions $\mathsf{N} \times \mathsf{M}$. Thus, the available species are equal to the dimension of the grid times the number of animals. The model has two kinds of rules: movement and interaction. The movement rules describe sheep trying to go into less dangerous cells, which have fewer wolves, while wolves tend to move to cells with a larger

number of sheep. For the interactions, we consider only a wolf eating a sheep. The probability of this event depends on the percentage of wolves and the probability of eating a sheep in the same cell. For example, we consider a scenario in which the predators are spawned at the vertexes of the grid and the prey at the centre. In this scenario, we start with a flock of 100 specimens at the centre of a grid of size $7 \times 7$, while 10 wolves are placed at each of the four angles. After a simulation with 100 time units, we observe that the mean number of sheep is drastically decreased, while the average number of wolves is the same, as shown in Figure 3.3. Moreover, we can observe that the spatial configuration is entirely different: the wolves are spread all over the grid.

```
const startW = 20; const startS = 100;
const N = 7; const M = 7;
...
species W of [0,N]*[0,M];
species S of [0,N]*[0,M];

rule go_up_S for i in [0,N] and j in [0,M] when j<M-1{...}
...
rule go_left_S for i in [0,N] and j in [0,M] when i>0{...}

rule eating for i in [0,N] and j in [0,M]{...}

rule go_up_W for i in [0,N] and j in [0,M] when j<M-1{...}
...
rule go_left_W for i in [0,N] and j in [0,M] when i>0{...}

...
system start_surrounded = W[0,0]<10>|W[6,0]<10>|W[0,6]<10>|W[6,6]<10>|S
    [3,3]<100>;
```

Listing 3.3: Predator-Prey model snippet in LPM

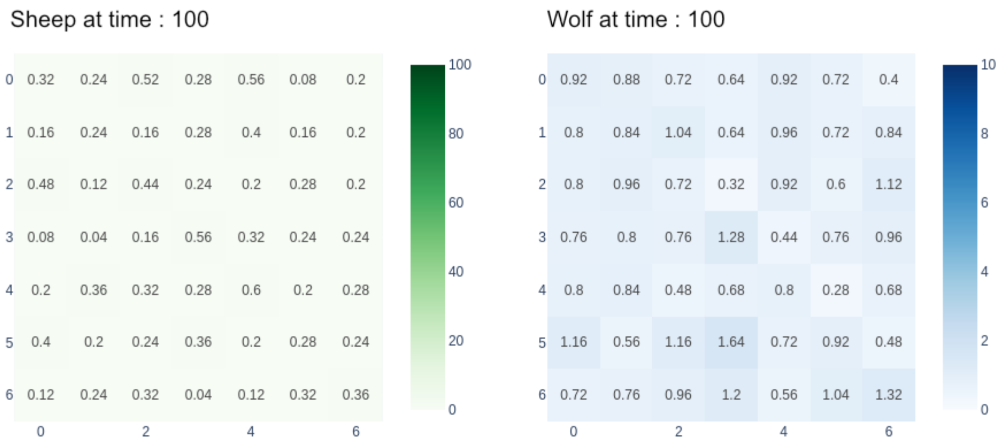



Figure 3.3: Distribution of Sheep and Wolves after 100 time units (10 runs)

# CHAPTER 4

# **YODA**: YET ANOTHER AGENT DESCRIPTION LANGUAGE

*If no mistake have you made,*

*yet losing you are...*

*a different game*

*you should play.*

An agent specification language is equipped with a set of *constructs* and *primitivies* that permit modelling a system according to a given *perspective*. In this chapter, the language YODA will be presented. YODA has been introduced first in [21] and further discussed in [22].

We first present the YODA computational model by focussing on three main concepts: the definition of agents, the definition of the environment where they operate, and how agents and environment interact. After that, the YODA specification language is presented. This is an *high-level* language that can describe MAS according to the YODA paradigm. Finally, we will show how, thanks to the integration of YODA in Sibilla, a simple scenario can be analysed.

A simple running example will be used throughout the chapter to introduce the YODA features and to help the reader understand how YODA works. We consider an example that is inspired by classical robotic scenarios where robots (the agents) move around in an environment to reach a goal area [25], as shown in Figure 4.1. In our scenario, robots can choose which direction to take autonomously, given a specific set of observations. These result from the interaction between an agent and its environment.

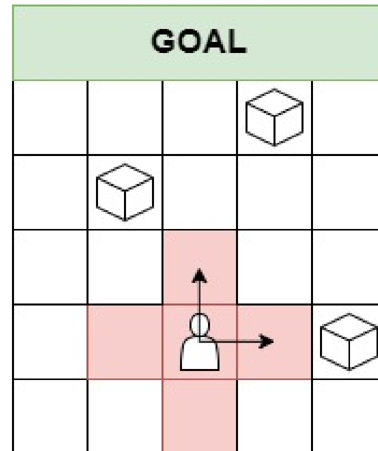



Figure 4.1: The basic Robotic Scenario

## 4.1 YODA Computational Model

The computational model of YODA permits to describe the behaviour of agents operating and evolving in an environment. In YODA, the environment is used to model the physical context where the agents operate. We define two types of dynamics, allowing the agent to evolve: *environmental* and *internal behaviour*. The environmental dynamic describes the interaction between an agent and its enclosing environment. For example, a typical use of the *environmental behaviour* is for modelling the agent *movement* over an area. The agent evolution is also influenced by its internal behaviour, depending on a *local knowledge* and on those data which are *sensed* from the enclosing environment.

As said in Section 2.1, YODA has been inspired by the definition of *Interpreted System* [104, 105] (IS). This approach has been used to reason about temporal and epistemic properties of agents [106]. An IS consists of a set of agents $A_1, \ldots, A_N$. Each agent $A_i$ has a *local state* $\ell_i \in L_i$ and can perform a finite set of *actions* $Act_i$. The *global state* of a system consists of a tuple $(\ell_1, \ldots, \ell_N)$ associating each agent $i$ with its local state $\ell_i$. At each step, each agent $i$ selects the following action to be performed according to the *local state* of $i$ and the *global states* of all the other agents.

Agents *live* in an environment modelled via a special agent, denoted by $\mathcal{E}$. Environment evolves similarly to the other agents and may affect their behaviour. At *global level*, a computational step is performed when each agent (including the environment) has selected and executed its action.

An IS is limited by how the environment is represented: the environment is considered as any other agent and does not permit appropriate modelling of the physical aspects of the system We explicitly model the environment in YODA to overcome this issue. The interactions between the environment and

agents are modelled in terms of *observations*. These are a class of *attributes* used to sense *data* from the environment. Like in IS, in YODA agents can perform actions selected according to the local agent state and the observations. The execution of an action triggers an update on the agent's local state.

### 4.1.1 YODA Agents

In YODA, each agent is represented via an agent name in $N \in \mathsf{AgName}$ and an *agent store* in $\sigma \in \Sigma$ that maps *attributes* in *values*. We let $\mathsf{Ag} = \mathsf{AgName} \times \Sigma$ denote the set of *YODA agents*. Elements of Ag are denoted by $A$, $A_1$, $A'$,. . . .

Each YODA agent provides three kinds of attributes: *conditions*, *features*, and *observations*. *Conditions* describe the set of attributes under the direct control of the agent. These represent the internal state of an agent and what is intrinsically known by it. If we consider our robotic running example, conditions can be used to model *robots' direction*. The robot can move according to the four cardinal points in this context. *Features* identify the environmental properties of an agent. These are the attributes under the control of the environment and change according to dynamics that may depend on *agents conditions*. Differently from the conditions, features are not readable nor updatable by the agent itself. In the running example, the *position coordinates* can be considered as features of a robot because it is out of the control of the agent. Finally, *observations* are used to perceive the enclosing environment. These attributes play the role of *sensors*. Observations are updated by the environment and then interpreted by the agent. The robots in the running example can observe their surroundings by sensing the presence of possible obstacles randomly placed in the area or understanding whether the agent has reached the goal area. According to these three elements, we let $\mathsf{Attr}_C$, $\mathsf{Attr}_F$ and $\mathsf{Attr}_\Omega$ indicate the set of *conditions*, *features*, and *observations*, respectively and define $\mathsf{Attr} = \mathsf{Attr}_C \uplus \mathsf{Attr}_F \uplus \mathsf{Attr}_\Omega$.

Let $\textsc{Val}$ denote the (measurable) set of values, an *agent store*, denoted by $\sigma \in \Sigma$, consists of a triple $(\sigma_c, \sigma_f, \sigma_\omega)$ where:

- $\sigma_c \in \Sigma_C : \mathsf{Attr}_C \to \textsc{Val}$ associates *conditions* to values;
- $\sigma_f \in \Sigma_F : \mathsf{Attr}_F \to \textsc{Val}$ associates *features* to values;
- $\sigma_\omega \in \Sigma_\Omega : \mathsf{Attr}_\Omega \to \textsc{Val}$ associates *observations* to values.

Each agent, at specific time intervals, performs an *action* (e.g. the robot moves towards a certain direction). Actions are identified by a name in ActName and represent a function that yields new agent conditions, given the agent conditions $\sigma_c$ and observations $\sigma_\omega$. An action can be performed only if it is allowed by the behaviour of an agent. This is defined in terms of two functions:

- $\beta \in \mathcal{B} = \Sigma_C \times \Sigma_\Omega \to Dist(\mathsf{ActName})$;
- $\alpha \in \mathcal{A} = \mathsf{ActName} \to \Sigma_C \times \Sigma_\Omega \to Dist(\Sigma_C)$.

The first function is directly related to the agent's behaviour. Given an agent's conditions and observations, it yields a probability distribution over action names that is used to select the next action to execute. For example, if the agent observes an obstacle, it avoids the object by turning in a specific direction. On the other hand, the second formula describes the effect of actions on the agent's conditions (producing a distribution of conditions).

An agent is defined in terms of its behaviour and actions. Thus, an agent definition $\Delta$ is a function that associates each agent name with its behaviour:

$$\Delta : \mathsf{AgName} \to \mathcal{B} \times \mathcal{A}$$

In the remainder of the section, for any $N \in \mathsf{AgName}$ such that $\Delta(N) = (\beta, \alpha)$, we will use $\Delta^N_\beta$ (resp. $\Delta^N_\alpha$) to denote the function $\beta$ (resp. $\alpha$).

### 4.1.2 YODA Environment

We describe an environment $\mathcal{E}$ as a function associating to each agent name $N \in \mathsf{AgName}$ two different functions:

- a *feature dynamic function* $\mathcal{F} : \Sigma_C \times \Sigma_F \to \Sigma_F$;
- an *observation function* $\Omega : \mathsf{Ag}^* \times \mathsf{Ag} \to Dist(\Sigma_\Omega)$.

The *feature dynamic function* describes how the features of an agent evolve in time. Given the current agent conditions $\sigma_c$ and features $\sigma_f$, $\mathcal{F}(\sigma_c, \sigma_f)$ yields agent features after one step. *Feature dynamic functions* are used to model aspects like the movement of agents, energy consumption and level of batteries. As we said, these aspects depend on the current agent conditions. For instance, an agent's position may depend on direction and speed, which the agent controls and stores as conditions.

The *observation function* is used to sense data from the environment. It takes the list sequence of the (other) agents in the system as arguments and returns the perceived observations. In our running scenario, observations are used to perceive obstacles and to check if the *goal area* has been reached. We remark that the result of the observation function is not a specific $\sigma_\omega$ but a probability distribution on $\Sigma_\Omega$ in order to model the data sensed from the environment are not exact and may be subject to errors.

For any environment $\mathcal{E}$ and $N \in \mathsf{AgName}$ such that $\mathcal{E}(N) = (\mathcal{F}, \Omega)$, we let $\mathcal{E}^N_\mathcal{F}$ (resp. $\mathcal{E}^N_\Omega$) to denote function $\mathcal{F}$ (resp. $\Omega$).

### 4.1.3 System Configuration

A *system configuration* consists of a sequence of agents. We let $S$ denote one element in $\mathsf{Ag}^*$. Each agent is identified using the index of its position in the sequence. For each $S \in \mathsf{Ag}^*$ we let $|S|$ denote the number of agents in $S$, while $S[i]$ denotes the agent in position $i$, where $1 \leqslant i \leqslant |S|$. Finally, for any $i$ such that $1 \leqslant i \leqslant |S|$, we let $S/i$ denote the configuration $S' \in \mathsf{Ag}^*$ obtained from $S$ by removing the agent in position $i$. Namely, the configuration such that $|S'| = |S| - 1$ and for any $j$ such $1 \leqslant j \leqslant |S| - 1$

$$S'[j] = \begin{cases} S[j] & (j < i) \\ S[j+1] & \text{otherwise} \end{cases}$$

### 4.1.4 YODA Semantics

Given an agent definition $\Delta$ and an environment $\mathcal{E}$, the behaviour of a YODA configurations are described in terms of a *Discrete Time Markov Chain* (DTMC) $\mathcal{M}^{\Delta}_{\mathcal{E}} = (\mathsf{Ag}^*, \mathcal{P}^{\Delta}_{\mathcal{E}})$. We recall that a Markov Chain is a stochastic process and describes a set of possible events that happen according to a given probability. This probability is influenced only by the state of the original event and not by previous iterations. In the case of a DTMC [107], the sequence of steps is defined in discrete time intervals. We can observe that under the assumption that VAL is measurable, the same can be considered for $\Sigma$ and $\mathsf{Ag}^*$. For the sake of simplicity, we do not provide all the details about the standard underlying probability settings. Moreover, we assume that VAL and Attr are countable. Interested reader can refer to [108, 109, 110] for details.

Given an agents definition $\Delta$ and an environment $\mathcal{E}$ the probability transition function $\mathcal{P}^{\Delta}_{\mathcal{E}} : \mathsf{Ag}^* \rightarrow Dist(\mathsf{Ag}^*)$, given a configuration $S$, gives the probability to jump to any other configuration:

$$\mathcal{P}^{\Delta}_{\mathcal{E}}(S) = \otimes_{i=1}^{|S|} \mathsf{ANext}^{\Delta}_{\mathcal{E}}(S/i, S[i])$$

where $\mathsf{ANext}^{\Delta}_{\mathcal{E}} : \mathsf{Ag}^* \times \mathsf{Ag} \rightarrow Dist(\mathsf{Ag})$ gives the probabilistic behaviour of an agent cooperating with a given tuple of agents, while $\otimes$ is used to compute the joint probability of the obtained probability distributions. We can observe that the position $i$ of an agent does not affect its behaviour.

Let $\pi_1, \pi_2 \in Dist(X)$, for some measurable set $X$, $\pi_1 \times \pi_2$ is the probability distribution $\pi' \in Dist(X \times X)$ such that $\pi'(a, b) = \pi_1(a) \cdot \pi_2(b)$, for any $a, b \in X$.

Function $\mathsf{ANext}^{\Delta}_{\mathcal{E}}$ describes the behaviour of an agent $A$ that is cooperating with a sequence of agents $S$. The result of all this function is a probability distribution on agents that measures the probability of an agent $A$ to evolve

in a step into the possible states. This probability distribution results from the computation of the following operations:

1. the environment updates agent's observations;

2. according to the current state and the new observations, the next action to execute is selected;

3. an action is executed to update the agent's conditions;

4. the environment updates the agent's features.

Considering our running scenario, the environment sends information about possible obstacles (or if they reached the goal area) to every agent as the first operation. Then, the robots decide which actions they could perform based on the new observations and their subsequent behaviour. Now, the robot can decide, for example, to turn in a certain direction and update its direction accordingly. Finally, the environment reads the new conditions and modifies each robot's position in the arena.

Let $N[\sigma_c, \sigma_f, \sigma_\omega]$ be an agent working in the configuration $S$. To obtain the measure of the possible observations, one has to apply the environment function $\mathcal{E}_\Omega^N(S, N[\sigma_c, \sigma_f, \sigma_\omega])$ that yields a probability distribution $\pi_\omega \in \Sigma_\Omega$.

Given $\sigma_c \in \Sigma_C$, $\pi_\omega \in Dist(\Sigma_\Omega)$ and $\beta : \Sigma_C \times \Sigma_\Omega \to Dist(\mathsf{ActName})$, we let $\mathsf{sampleAction}(\beta, \sigma_c, \pi_\omega)$ denote the probability distribution $\pi$ over $\mathsf{ActName} \times \Sigma_\Omega$ such that:

$$\pi(a, \sigma_\omega) = \beta(\sigma_c, \sigma_\omega)(a) \cdot \pi_\omega(\sigma_\omega)$$

Function sampleAction yields the probability distribution associating the following action to execute with specific observations.

To compute the execution results of actions, function applyAction is used. This function takes as input an action execution function in $\mathcal{A}$, agent's conditions $\sigma_c$ and a distribution $\pi \in Dist(\mathsf{ActName} \times \Sigma_\Omega)$ and yields a probability distribution in $Dist(\Sigma_C, \Sigma_\Omega)$ that measures the probability that the agent evolves in a state with the given conditions and observations. Namely, $\mathsf{applyAction}(\alpha, \sigma_c, \pi) = \pi_s$ where:

$$\pi_s(\sigma'_c, \sigma_\omega) = \sum_{a \in \mathsf{ActName}} \alpha(a)(\sigma_c, \sigma_\omega)(\sigma'_c) \cdot \pi(a, \sigma_\omega)$$

Given a probability distribution on $\pi \in Dist(\Sigma_C, \Sigma_\Omega)$, we can finally apply features dynamics $\mathcal{F}$ to obtain the probability distribution of possible agent configurations. We let $\mathsf{computeFeatures}(N, \mathcal{F}, \sigma_f, \pi)$ give the probability distribution $\pi_{\mathsf{Ag}} \in Dist(\mathsf{Ag})$ such that:

$$\pi_{\mathsf{Ag}}(M[\sigma'_c, \sigma'_f, \sigma'_\omega]) = \begin{cases} \pi(\sigma'_c, \sigma'_\omega) & M = N \wedge \sigma'_f = \mathcal{F}(\sigma_f, \sigma'_c) \\ 0 & \text{otherwise} \end{cases}$$

Summing up, given $S \in \mathsf{Ag}^*$ and $N[\sigma_c, \sigma_f, \sigma_\omega] \in \mathsf{AgName}$, function $\mathsf{ANext}_{\mathcal{E}}^{\Delta}$ can be defined as follow:

$$
\begin{aligned}
&\mathsf{ANext}_{\mathcal{E}}^{\Delta}(S, N[\sigma_c, \sigma_f, \sigma_\omega]) = \\
&\qquad \mathsf{let}\ \pi_a = \mathsf{sampleAction}(\Delta_{\beta}^{N}, \sigma_c, \mathcal{E}_{\Omega}^{N}(S, N[\sigma_c, \sigma_f, \sigma_\omega]) \\
&\qquad \mathsf{in} \\
&\qquad\quad \mathsf{computeFeatures}(N, \mathcal{E}_{\mathcal{F}}^{N}, \sigma_f, \mathsf{applyAction}(\Delta_{\alpha}^{N}, \sigma_c, \pi_a)))
\end{aligned}
$$

## 4.2 YODA Specification Language

We saw how YODA allows us to describe MASs. However, to fully exploit the functionalities of formalism, we need to introduce a specification language. This section provides an insight into how we can use YODA *Specification Language* to model Multi-Agent Systems. We focus on the process of designing models, explaining how and why we write certain rules. It should be noted that the syntax presented is shortened, with respect to the actual grammar. The interested reader can find the full specification grammar in the Sibilla repository[1] and in Appendix B of this document. To help the designer in specifying models with YODA, a light language extension for the Visual Studio Code editor is available on GitHub as well[2]. It provides a set of rules for syntax highlighting and snippets to ease the task of writing a YODA specification.

According to the formalism seen in the previous section, we need to consider three main components to define a YODA model: *agents*, *systems*, and *configurations*. The developed grammar has been conceived trying to keep it as simple as possible: the model specification, the actions, and the behaviour are easy to understand. However, simplicity comes with a price, and the more complex the system to be developed, the more effort the designer should put into the model specification and the description of how the system evolves. In detail, a YODA specification consists of a list of elements

- Constant, Parameters, and Functions
- Agents, Scene Elements, and Groups
- Environment
- System Configurations
- Measures and Predicates

[1]The tool is available at `https://github.com/quasylab/sibilla`

[2]The extension is available at `https://github.com/quasylab/LangExt4Sibilla`

Below each of them is presented by using code snippets and the considered running example.

### 4.2.1 Parameters, Constants, and Functions

In the previous section, where we described the formalism, we stressed that values play a central role in YODA, thus we need to include *constants* and *parameters*. These elements allow us to associate names with values and to reuse them in different parts of the model. As one can imagine, we consider constants as fixed values that can not be changed. On the contrary, parameter variables can be changed at runtime. We will see later that this is useful when one is interested in verifying the behaviour of a system under different conditions, such as the number of agents or the number of obstacles in the arena.

Syntax of constants and parameters declarations is the following:

**param** <*name*> **=** <*expr*> ;
**const** <*name*> **=** <*expr*> ;

where <*name*> refers to the syntactic category of *names*, while <*expr*> is used when defining an expression. A valid *name* is a sequence of letters, digits or the symbol '_' that does not start with a digit, similar to many other programming languages. On the other hand, expressions are built by using standard boolean/arithmetic operators.

In YODA, three basic types can be used: `int`, `real` and `bool`. These refer to integer values, floating point values in double precision, and boolean values. Other data types could be easily integrated into our language. However, for the sake of simplicity, we have limited our attention to those commonly used in many models. The type associated with constants and parameters is directly inferred from the used expression.

**Example 1** (Parameters and Constants)**.** *In the case of our robotic scenario, we will use constants to define the dimensions of the grid where the agents operate (Listing 4.1). These are defined by the constants* `height` *and the* `length`*. On the contrary, we use parameters to define the number of obstacles* `no` *and the number of robots* `na`*. Following this approach, we can analyse our model by varying the number of agents without changing the configurations.*

```
const height = 10;
const length = 10;
param no = 10;
param na = 5;
```

Listing 4.1: Parameters and constants variables in a Robotic Scenario

If we need to perform elaborate computations, we can include in a YODA specification new *functions*. This can improve the readability of the code by reducing the lines of code and modularising the code. Syntax of function declarations is the following:

**let** <*name*> ((<*type*> <*name*>)*) =
<*functionStatement*>
**end**

A function is characterised by a name and a (possibly empty) list of arguments. Following a functional approach, the body of a function can be a `return`, an `if then else`, or a `let in` statement.

Finally, any expression can only use previously defined constants, parameters and functions. This avoids recursive computations that may lead to divergent behaviour.

**Example 2** (Functions)**.** *The following is the definition of a function that can be used to compute the distance between two points* $(x_1, y_1)$ *and* $(x_2, y_2)$*:*

```
let distance(real x1, real y1, real x2, real y2) =
    let dx = x1−x2
    and dy = y1−y2
    in
        return sqrt(pow(dx,2)+pow(dy,2))
end
```

Listing 4.2: Function in a Robotic Scenario

### 4.2.2 Agents and Scene Elements

In a YODA specification, the *agent* is divided into five major components: *conditions*, *features*, *observations*, *actions*, and *behaviour*. The first three components are used to store agent attributes. The first, `conditions`, declares internal attributes, such as a chosen direction or battery charge percentage, and expresses what is intrinsically known by the agent. On the other hand, external information that the agent can not modify is included in the `features` declaration, like the position on a grid. Finally, the `observation` attributes store data about the agent's surroundings. These can be, for example, observations about the occupancy of a certain space or the type of tile the agent is currently on.

These three categories of agent attributes can be described in the following manner:

**agent** <*name*> =
  **conditions**:
    (<*type*> <*name*> = <*expr*>;)*
  **features**:
    (<*type*> <*name*> = <*expr*>;)*
  **observations**:
    (<*type*> <*name*> = <*expr*>;)*
...

**Example 3** (Robot Attributes)**.** *In our model, each robot is represented with an agent named `Robot`. For the sake of simplicity, we assume that our robots move on a grid. At each step, the direction is selected depending on the perceived obstacles. Indeed, each robot is equipped with four sensors that permit checking if an obstacle is located in front of one of the four possible directions (*north, south, east *and* west*). Moreover, a further sensor is used to check if the* goal area *has been reached. The first part of the declaration of the agent Robot is reported in Listing 4.3.*

```
agent Robot =
  conditions:
    int dirx = 0;
    int diry = 0;
  features:
    int x = 0;
    int y = 0;
  observations:
    bool north = false;
    bool south = false;
    bool east = false;
    bool west = false;
    bool goal = false;
  ...
```

Listing 4.3: Attributes of a Robot Agent

*To represent the Robot's direction, integer conditions `dirx` and `diry` are used, while the features `x` and `y` are used to store its position. Observations contain the declaration of boolean attributes `north`, `south`, `east` and `west`, that perceive obstacles in the forth directions, and the attribute `goal` that checks if the goal area has been reached.*

In the previous section, when describing the actions formalism, we saw how these are used to update the agent's conditions. For example, we can declare an action that allows the agent to know if it is going to accelerate in a specific direction. The declaration of possible actions is contained in a block starting with the keyword `actions`. The whole block syntax is reported below:

**actions**:
  <*name*> [ <*attrUpdate*>* ]

An action can be identified using a unique $< name >$. We associate a sequence of commands for each action name that uploads the agent condition attributes. Each update is referred to as $< attrUpdate >$. We can have different attribute update forms, which can be seen in the full grammar. If the action body has no $< attrUpdate >$, the agent will move to the next step without updating anything. Here we mention the simple update that permits changing the value of a specific attribute:

$$<name> \text{ <- } <expr>$$

In the above update, $<name>$ must be an agent condition variable, while $<expr>$ can only refer to conditions and observations.

**Example 4** (Robot Actions)**.** *The robot performs actions (declared in Listing 4.4) to reach the goal area and avoid obstacles on the grid. The actions modify the conditions of the agent, namely changing the values of attributes* `dirx` *and* `diry`*. We can also add a stop action that will be performed when the robot reaches the goal area and does not need further operations.*

```
...
actions :
  moveNorth [ dirx <- 0; diry <- 1;]
  moveSouth [ dirx <- 0; diry <- -1;]
  moveEast [ dirx <- 1; diry <- 0;]
  moveWest [ dirx <- -1; diry <- 0;]
  stop [ dirx <- 0; diry <- 0; ]
...
```

Listing 4.4: Actions of a Robot Agent

Given the specification of the attributes and the actions, the final component that should be declared while designing a YODA agent is included in the block `behaviour`. This block describes how the agent will act given a certain guard condition. In fact, it consists of a sequence of *guards* associated with a list of *weighted action names*. The latter permits associating action names with weights and is used to derive a probability distribution among actions. The syntax of the behavioural block is presented below:

**behaviour**:
 **when** $<guardExpr>$ -> $[(<weightedActions>)]$
 (**orwhen** $<guardExpr>$ -> $[(<weightedActions>)])*$
 **otherwise** $[(<weightedActions>)]$
**end**

The first satisfied guard is found to select the action to execute, and the corresponding action probability is generated. If no guard is satisfied, the probability in the case `otherwise` is returned.

**Example 5** (Robot Behaviour)**.** *The robot's behaviour, declared in Listing 4.5 is based on the observations about its surroundings. It will stop moving if it has reached the goal area because the objective has been achieved. The movement is performed according to the presence of obstacles around the agent. First, the agent observes if there is something in the box in front of itself. In the negative case, the robot will move upwards; otherwise, it will decide in a non-deterministic fashion whether to turn left or right. The other two behavioural rules allow the robot to not crush into the wall or an obstacle. Indeed if the agent finds an obstacle in the* `north` *and in the* `east`*, it will turn left. Similarly, the agent will move right if the second obstacle is on the* `west`*. The agent will automatically stop moving if none of the precedent conditions are matched.*

```
  ...
  behaviour :
    when goal -> [ stop: 1; ]
    orwhen !north -> [ moveNorth: 1; ]
    orwhen north -> [ moveEast: 0.5; moveWest: 0.5; ]
    orwhen north && east -> [ moveWest: 1; ]
    orwhen north && west -> [ moveEast: 1; ]
    otherwise [ stop: 1; ]
end
```

Listing 4.5: Behaviour of a Robot Agent

Sometimes, it is helpful to consider **static** agents. In this case, one can use *scene elements*. These are static objects that other agents can perceive. *Scene elements* and are defined in the following manner:

**element** <*name*> =
<*type*> <*name*> = <*expr*>;
...
<*type*> <*name*> = <*expr*>;
**end**

A scene element consists of a set of (static) attributes associated with a default value. For each attribute, we declare the <*type*> (*integer*, *real*, and *boolean*), an identifier <*name*>, and a (default) value obtained by the evaluation of the given expression <*expr*> that should be consistent with the assigned type.

**Example 6** (Obstacles)**.** *The robots must avoid obstacles randomly placed in the operational area to complete their tasks. The obstacles are scene elements, declared in Listing 4.6, that require a set of coordinates. They will be initialised later.*

```
element Obstacle =
  int posx = 0;
  int posy = 0;
end
```

Listing 4.6: Obstacles in a Robotic Scenario

### 4.2.3 Environment

A YODA system is defined in the syntax as the entity that regulates the whole enclosing environment. Specifying a system requires the programmer to consider the *sensing function* and the *dynamics of the environment*. The sensing function is crucial to updating every available agent's observations. In this component, we need to identify the type of agent, i.e. the name we gave in the previous declarations, that will be updated. Here, we can only modify the observations fields using a group of <*attrUpdate*> constructs, already discussed in the block `actions` presented above. Similarly, we can update the agent information state in the environment dynamics declaration using the same construct of the sensing function.

```
environment:
    sensing:
        (<agentName> [
            <attrUpdate>
        ])*
    dynamic:
        (<agentName> [
            <attrUpdate>
        ])*
end
```

In the context of sensing block, *group expressions* can be used. These expressions permit collecting data from a group of agents and allow to:

- check if there exists an agent belonging to a specific class satisfying a given boolean expression:

  **any** (<*name*>)? ':' <*expr*>

- count the number of agents belonging to a specific group whose attributes satisfy a given predicate:

  **#** (<*name*>)? '['<*expr*> ']'

- evaluate the *min*, *max* and *sum* and *mean* of an expression computed over the attributes of the agents belonging to a specific group whose attributes satisfy a given predicate:

**min** (<*name*>)? '['<*expr*> ']'. expr
**max** (<*name*>)? '['<*expr*> ']'. expr
**sum** (<*name*>)? '['<*expr*> ']'. expr
**mean** (<*name*>)? '['<*expr*> ']'. expr

These group expressions permit modelling stigmergic behaviour [111]. Each group expression may contain a predicate (a boolean expression < *expr* >), which filters those entities that match the addressed conditions. For example, in the *any* group expression, we consider any agent that passes the requested filter.

To compute these expressions we may need to relate the values of attributes in a specific agent, with the agents located in the surrounding. In this case, the prefix `it.` is used to retrieve attributes from the agent where the expression is evaluated:

**it** . <*name*>

**Example 7** (Robotic Scenario Environment)**.** *The environment declaration (Listing 4.7) includes the functions to update the agents' observations and features. In the first case, we exploit the group expression* `any` *to check if an obstacle is located in a particular position. In case we are searching for an obstacle in the north, we look at the position above the agent using the* itself *expression (*`it.x, it.y+1`*) and we compare it with every initialised obstacle until we find one that is in the same position. Indeed the* `any` *group expression returns a boolean value, as required by the* `north` *observation attribute. The same principle applies to the other three observation directions. Regarding the* `goal` *observation attribute, the environment just checks if the value of* `it.y` *is equal to the height of the grid plus 1.*

*For the feature updates, the environment simply updates the two coordinates according to the new condition attribute values. In this case, we don't need to use the* `it.` *expression because the operation is done on local variables. The actual position coordinates are summed with the new direction values.*

### 4.2.4 Configuration Declaration

While declaring a model in YODA, designers need to consider the system's initial configuration. This construct is used to initialise the agents and the elements that will be loaded in the environment. A configuration requires an

```
environment:
  sensing :
    Robot [
      north <- any Obstacle : (posy==it.y+1)&&(posx==it.x);
      south <- any Obstacle : (posy==it.y-1)&&(posx==it.x);
      east <- any Obstacle : (posx==it.x+1)&&(posy==it.y);
      west <- any Obstacle : (posx==it.x-1)&&(posy==it.y);
      goal <- it.y==height+1;
    ]

  dynamic :
    Robot [
      x <- x + dirx;
      y <- y + diry;
    ]
 end
```

Listing 4.7: Environment in a Robotic Scenario

identifier that the simulator will use to initialise data. Then, we must declare both agents, scene elements, and initial attributes. If attributes are not initialised, the default value defined in the declaration will be used. Iterative and conditional operators can be used to instantiate a multitude of agents.

**configuration** $<confName>$:
$<collectiveExpressions>$
**end**

The syntactic category of $<collectiveExpressions>$ allows us to populate the system with an arbitrary number of agents and scene elements. Three different blocks can be used:

- instantiating an agent with a set of given attribute values:

  $$<name>\ \text{'['}\ (<name>\ \text{'='}\ <expr>\ \text{';'})^*\ \text{']'}$$

  In this case, we need only to indicate the name of the agent or scene element to instantiate together with the (optional) assignment of attributes.

- iterating through a set of values:

  **for** $<name>$ $<setOfValues>$
  **do** $<collectiveExpressions>$
  **endfor**

  The *for-cycle* statement permits simplifying the instantiation of a group of agents. The group of values can be selected from an interval, an enumeration, or a random expression. In the latter case, one can also impose that randomly selected values can be distinct. We will use this feature later when we place agents/elements in distinct positions.

- using a ternary if block to differentiate an agent depending on a guard:

**if** $<expr>$
**then** $<collectiveExpressions>$
**else** $<collectiveExpressions>$
**endif**

**Example 8** (System Configuration)**.** *The initial configuration code for the Robotic Scenario, shown in Listing 4.8 defines the initial position of both agents and obstacles. In both cases, we cycle through a number of steps equal to the corresponding parameter (*`na` *represents the number of agents,* `no` *the number of obstacles) and we obtain an integer value from the* `floor(U[n, m])` *function, which calculates the floor approximation of a random value between* `0` *and the* `width`*. The two entities differ because, as stated before, the agents start at the bottom row, while the obstacle can be randomly located on any column and from the second row.*

```
configuration Main:
  for i sampled distinct na time from floor(U[0, width]) do
    Robot[ x = i ; y = 0 ; ]
  endfor
  for o sampled distinct no time from floor(U[0, width]) do
    Obstacle[ posx = o; posy = floor(U[2, height]); ]
  endfor
end
```

Listing 4.8: Configuration of a Robotic Scenario

### 4.2.5 Measures

A *measure* is a function associating a real value to YODA configurations. Measures are used during simulations to collect data and to analyse system behaviour.

The syntax of a measure declaration is the following:

**measure** $<name> = <expr>$

The expression associated with a measure can contain one of the *group expressions* described above.

**Example 9** (Robot's Measures)**.** *In our robotic scenario, one could be interested in measuring the number of robots that have reached the* goal area*. In Listing 4.9, we show how to declare a measure for our robots.*

```
measure inGoalArea = #Robot[it.goal];
```

Listing 4.9: Measure of a Robotic Scenario

### 4.2.6 Predicates

A predicate is a boolean expression that can be evaluated either at a global level, i.e. by taking into account the properties of a YODA configuration, or at a local level, by considering the properties of a single agent. In the first case, we will speak about *global predicates*, and in the latter, about *local predicates*.

To declare a predicate, the following syntax can be used:

$$(\textbf{local})?\ \textbf{predicate}\ <name> = <expr>$$

The declared predicate is *global* if the optional keyword `local` is omitted, *local* otherwise. The expression associated with a predicate is a boolean expression. When the predicate is global, this expression can contain one of the *group expressions* outlined above.

**Example 10** (Robot's Predicate)**.** *Global predicates can be used to identify specific situations in our system. For instance, in the case of our robotic scenario, we can check if all the robots have reached the goal area. This is presented in Listing 4.10.*

```
predicate success = %Robot[it.goal] == 1;
```

Listing 4.10: Predicate of a Robotic Scenario

## 4.3 Tracing and Simulating YODA with Sibilla

To support simulation and analysis of YODA models, our formalism has been integrated in Sibilla. The integration consists in a *module* that provides:

- a set of classes implementing the computational model described in Section 4.1;
- a parser that loads a specification file as described in Section 4.2, checks its correctness, and instantiates the model for the analyses.

Going in-depth with the description of these classes goes outside the interests of this thesis. Indeed, they consist of the implementation of the semantic functions introduced in Section 4.1 and of the implementation of

standard parsing techniques. An interested reader can get more details about the implementation by looking at the Sibilla Git repository accessible via the following link: `https://github.com/quasylab/sibilla`.

In the rest of this section, we will show how Sibilla can be used to trace and simulate YODA specifications.

### 4.3.1 Tracing the Running Scenario

Sibilla permits tracing models' behaviour to render the sequence of steps performed by the agents in the system. YODA exploits this functionality of Sibilla to get a complete insight into a modelled scenario, allowing the user to understand the evolution of each agent. As previously said in Section 3.2, we must define the tracing specification in a file describing the set of features collected from the agents.

```
shape = square;
colour = {
    when goal: green;
    otherwise: red;
}
```

Listing 4.11: Tracing Specification for a Robotic Scenario

In the case of our running scenario, we can consider the rules declared in Listing 4.11. These are relatively simple. A square represents each agent, green-coloured if the goal area has been reached and red otherwise. Agents are plotted in a 2D area. The `x` and `y` feature attributes are omitted because they match the tracing elements. Finally, the `direction` is set to a default value because it is neither specified nor implicitly included in the model.

We can execute the simulation on our Robotic Scenario and obtain its traces. Using the Sibilla native shell, we type down the commands in Listing 4.12.

```
module "yoda"                                  /* Load YODA model */
load "robotic.yoda"                            /* Load system specification */
init "Main"                                    /* Set initial configuration */
deadline 100                                   /* Set simulation deadline */
trace "robotic.trc" in "./results" h = true    /* Trace and save results */
```

Listing 4.12: List of commands to start simulating the Robotic Scenario

Listing 4.12 shows the commands for simulating the running example. The first command, `module "yoda"`, asks the runtime to use the YODA module as an interpreter for the specified model, which is loaded in the second line with the `load` command by identifying the path of the file. To initialise the model, we use the `init` command followed by the configuration name,

which should be contained in the specification. The `deadline` command is the time horizon of the simulation. Finally, with the command `trace`, we save the results of a single simulation: this command needs the trace specification we saw earlier, the output folder, and the header flag (to add the headers to the .csv files).

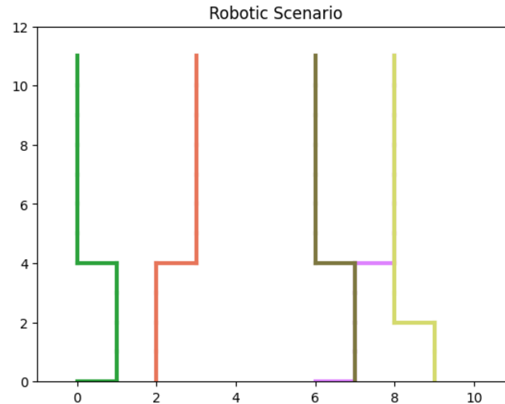



Figure 4.2: Simulation Results of a Robotic Scenario

Figure 4.2 presents a 2D static plot of five robot trajectories. The plot reports the behaviour of five robots. Following a probabilistic approach, each turns left or right when encountering an obstacle. In the running scenario, collisions are not considered. Indeed, the traces of the three agents on the right side of the picture overlap.

### 4.3.2 Simulating the Running Scenario

Tracing permits getting information about a single simulation run. However, to infer performance measures, a larger number of simulations is needed for a more detailed statistical analysis. Sibilla has functionalities for collecting and aggregating simulation results. This section exploits these functionalities to provide new insight into the robotic scenario. We focus on the execution of two different analyses: *First-Passage-Time* (FPT) and *Reachability*.

The FPT analysis [112] determines the time an observed variable takes to reach a certain state. This kind of analysis is usually done in economic cases to represent bankruptcy when a model reaches the value of 0 or in medicine to represent patients' issues. We refer to the Reachability [113] analysis as the ability of a distributed system to reach certain global states.

As we saw in Section 4.2.6, we use a predicate to consider the fraction of agents that have reached the `goal` area. The predicate is satisfied if the fraction equals 1, meaning that every agent has completed its task. To improve the results of both analysed properties, we expand the considered system

by increasing the number of agents and obstacles and enlarging the arena. We use an arbitrary value `k` to modify `no`, the number of obstacles, and the `height` and `width` of the arena. We initialise 20 agents: a higher number will only slow down the simulation without producing any meaningful results. Concerning the simulation parameters, we set the deadline at `k+20` time units and execute 100 replicas.

For the First-Passage-Time analysis, we want to know the mean time it takes for the system to satisfy the predicate. Due to the system's simplicity, where the robots only turn right or left and we have only one obstacle for each column, we don't expect to satisfy the property before a time span equal to an arbitrary value `k+1`. This "perfect" case means that every agent encounters only one obstacle, which is impossible. In Figure 4.3, we present the results of our analysis.

| K | Test | Hits | Mean | SD | Min | Q1 | Q2 | Q3 | Max |
|---|---|---|---|---|---|---|---|---|---|
| 20 | 100 | 100 | 25.83 | 2.15 | 24 | 25 | 25 | 26 | 40 |
| 50 | 100 | 100 | 55.5 | 1.55 | 54 | 55 | 55 | 56 | 65 |
| 100 | 100 | 100 | 105.32 | 1.1 | 104 | 105 | 105 | 106 | 110 |
| 200 | 100 | 100 | 205.18 | 0.8 | 204 | 205 | 205 | 206 | 208 |

Figure 4.3: FPT Analysis Results for the Robotic Scenario

We consider four scenarios, with `k` set to 20, 50, 100, and 200 respectively. The *Mean* time to satisfy the predicate requires around 5 time units more than `k`, with a more precise result for each case study, as shown by the decreasing *Standard Deviation*. The system requires always at least 4 time units more than the `k`. Instead, the *Max* time is lesser with a bigger `k`.

A reachability property shows the probability of a system satisfying a predicate within a deadline. Figure 4.4 shows the results of the Robotic Scenario where `k` is set to 20. Our reachability analysis of the Robotic Scenario shows that the predicate is unlikely to be satisfied after 20 time units. The probability of having a successful model is significant around the 24th time interval. The probability of seeing that every agent has reached the goal area after 25 time units. The curve stabilises itself around the 40th interval. These observations confirm what we saw in the FPT analysis, thus the model is verified.

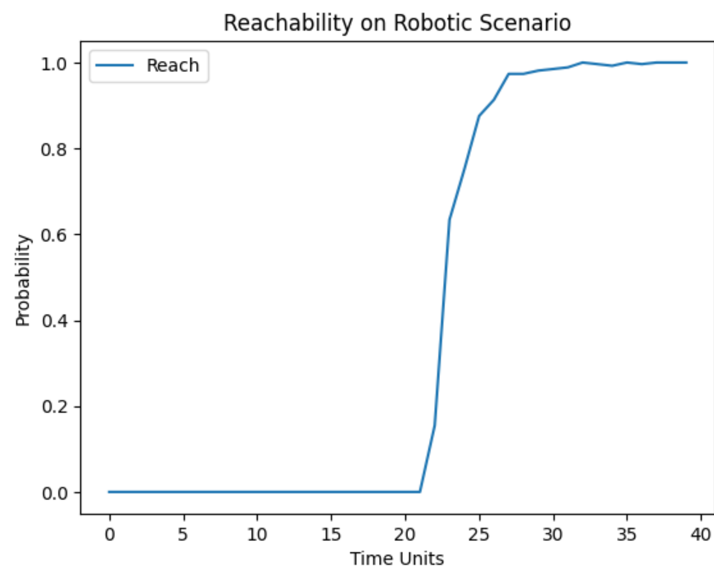



Figure 4.4: Reachability Analysis Results for the Robotic Scenario

# CHAPTER 5

# YODA AT WORK

*Truly wonderful*
*the mind of a child is.*

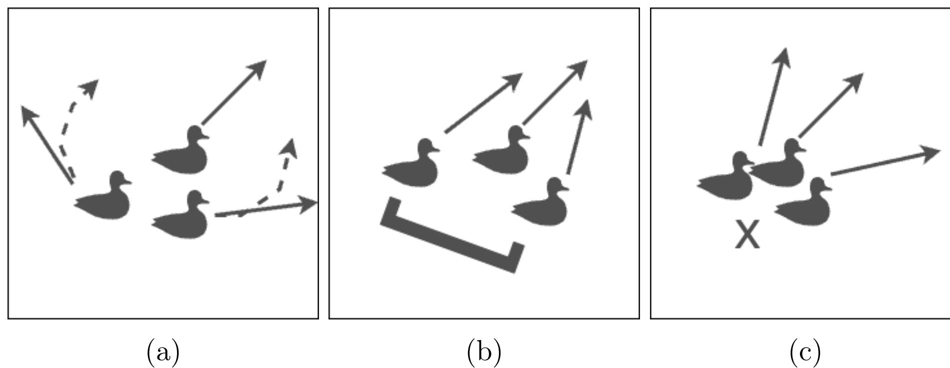
Multi-Agent Systems vary in composition, finality and objectives. Understanding their intricacies and possible limits is complicated without proper models and simulations. For these reasons, we decided to design a set of case studies that embrace the new language and use them to understand if the YODA may be a viable framework to simulate MASs.

This chapter contains four additional examples taken from the literature, that have been redefined according to YODA grammar rules. The first example, in Section 5.1 is based on a *flock* scenario, which considers the coordinated movement of a group of birds.. The second example presented in Section 5.2 describes a team of robots (or more generally agents) aiming to *find* an object, for example, an injured one or an infesting grass randomly located on the map. In Section 5.3, the third example shows how we can simulate a *SEIR*, commonly known as an epidemic scenario. Finally, a consensus-based example called *Red-Blue* scenario is presented in Section 5.4, to mimic the interactions between two types of agents. Differently from the *Robotic Scenario*, the examples considered in this section allow us to appreciate how the YODA framework permits modelling and analysing the behaviour of a collective of interacting agents.

## 5.1 Flock

A flock is considered a group of birds (or boids) freely roaming around. Originally, this model was developed by Reynolds in [26], but it was further

explored and improved, for example, in [114]. These agents move in a certain direction, given an acceleration, from an origin point. However, birds adopt three types of policy to avoid collisions: alignment, cohesion, or separation. In the first case (Figure 5.1a), each agent aligns itself according to its neighbours' direction. When adopting the cohesion (Figure 5.1b), a bird tries to keep a short distance from each other, meaning that it will estimate the future position of its neighbour and turn towards it. Finally, in the separation policy (Figure 5.1c), birds try to avoid colliding by distancing themselves.




(a) (b) (c)

Figure 5.1: The three policies of a flock

The flock behaviour used here is a reinterpretation of Reynolds's original model. First of all, the birds are free to move without caring about possible obstacles in the environment. Then, the bird movement is based on a mean point calculated on the mean position of every available agent. The decision is taken considering two main conditions: the position and the distance. Regarding the first condition, each agent is informed about the actual position of the mean point concerning its position. In the other case, the bird observes the distance from said point. After these two variables are observed, the agent decides whether to distance itself, get closer, or maintain its direction. For simplicity reasons in reading the model, we omitted the steering factor of the bird, meaning that the bird immediately turns in the chosen direction.

### 5.1.1 Flock Specification

The Flock scenario specification is defined in Listing 5.1. Unlike the other two case studies, we will not consider the presence of obstacles or inanimate objects, thus no elements are declared. We declare five parameters for this model: the number of birds available in the simulation `nbirds`, the maximum distance `maxDist`, the minimum one `minDist`, the actual speed of the bird `speed`, and finally the support parameter `delta`.

An agent `Bird` is equipped with the three main sets of attributes: *conditions*, *features*, and *observations* For what concerns the first attribute store, we only need to consider the `angle`, which is initialised with a random value in radians between 0 and $\pi$. As features, we consider the coordinates `x` and `y`. Finally, the bird can observe where the `centerOfMass` is oriented in respect of the agent and if it `isTooClose` or `isTooFar` from said focal point. Regarding the available actions, an agent can only perform two custom actions. The first one is `return`, which allows the agent to reach for the centre of mass by modifying the `angle` with the perceived `centerOfMass` summed with a random value between $\pi/4$ and $-\pi/4$ radians. Similarly, the `escape` action considers the focal point of the flock, but the added random value is between $3\pi/4$ and $5\pi/4$ radians, which permits the agent to move away. Finally, the agent's behaviour is expressed via three rules based on the distance from the mass centre. We highlight that the default rule, identified by the keyword `otherwise`, performs the action `skip`: this is a static action that does not require any specification and leaves unchanged the state attributes.

The environment updates the observations and the position as well. To compute the observations, we exploit *group expressions*, that calculate a value on a defined group of agents, and *temporal variables*. The latter are used to update each of the three bird observations. The position is updated in the `dynamic` by adding to the actual position the sine or cosine of the `angle` multiplied by the `speed`.

### 5.1.2 Flock Simulation

The configuration seen in Listing 5.2 is used to initialise the birds. We position a number of bird (`nbirds`) in random coordinates using a single for-cycle, which calculate both the x-coordinate and the y-coordinate.

For this model, we use the trace shown in Listing 5.3. Similarly to the Robotic Scenario, we don't explicitly declare the coordinates, which are directly taken from the flock specification. For this simulation, we used the z-axis instead of the y-axis because it will be read by the visualiser proposed in the next chapter. The angle variable seen in the specification is assigned to the direction field. The shape is associated with a bird and the colour is based on the variables `isTooFar` and `isTooClose`.

Considering the proposed model, we expect the flock dimension to expand and shrink. In fact, we do not consider the steering phase for this version of the model, and birds can turn directly in the assigned direction. Using Figures 5.2 (three agents), 5.3 (five agents), and 5.4 (ten agents), we discuss the results obtained from the simulation. In the plotted results, each bird of the flock is represented by a different random colour. Moreover, we set the deadline for the simulation to 100 time units.

The scenario where we consider three birds allows us to better appreciate

```
param nbirds = 5;
param maxDist = 2; param minDist = 1;
param speed = 1; param delta = 2;

agent Bird =
  state :
    real angle = 0.0;
  features:
    real x = 0.0;
    real y = 0.0;
  observations:
    real centerOfMass = 0.0;
    bool isTooFar = false;
    bool isTooClose = false;
  actions:
    goBack [angle <- centerOfMass+U[-1/4*PI, 1/4*PI];]
    escape [angle <- centerOfMass+U[3/4*PI, 5/4*PI];]
    skip [angle<-angle;]
  behaviour:
    when isTooFar -> [goBack:1;]
    orwhen isTooClose -> [escape:1;]
    otherwise [ skip : 1;]
end

environment :
  sensing:
    Bird [
      let meanX = mean Bird[distance(it.x, it.y, x, y)<delta]. x
      and meanY = mean Bird[distance(it.x, it.y, x, y)<delta]. y
      in
        isTooFar <- distance(it.x, it.y, meanX, meanY) > maxDist;
        isTooClose <- distance(it.x, it.y, meanX, meanY) < minDist;
        centerOfMass <- angleOf(meanX, meanY, it.x, it.y );
      endlet
    ]
  dynamic:
    Bird [
      x <- x + cos(angle)*speed;
      y <- y + sin(angle)*speed;
    ]
end
```

Listing 5.1: Flock Scenario Specification

```
configuration Main :
  for o sampled distinct nbirds time from U[0, nbirds] do
    Bird [ x = o; y = o; ]
  endfor
end
```

Listing 5.2: Flock Scenario Configuration

```
direction = angle;
shape = bird;
colour = {
    when isTooFar: blue;
    when isTooClose: red;
    otherwise: yellow;
}
```

Listing 5.3: Tracing Specification for a Flock

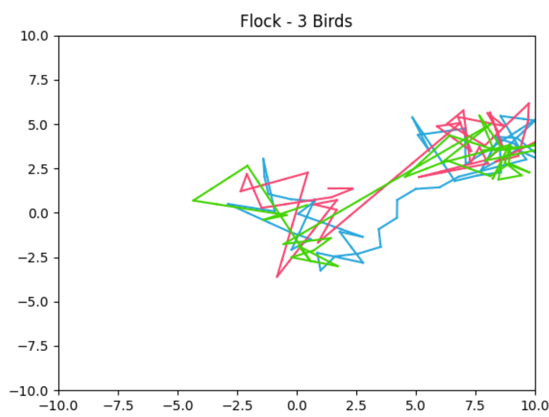



Figure 5.2: 3 birds flock

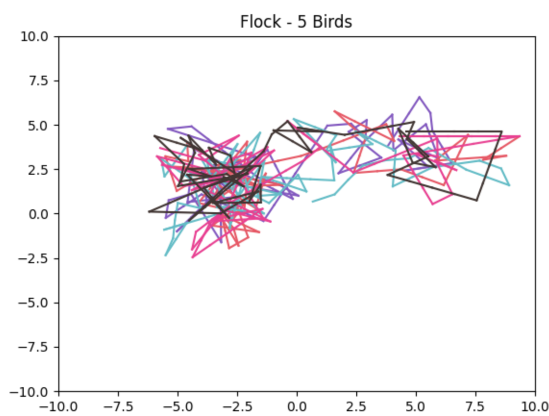



Figure 5.3: 5 birds flock

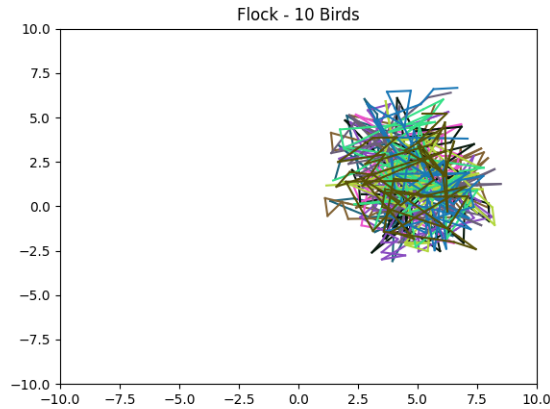
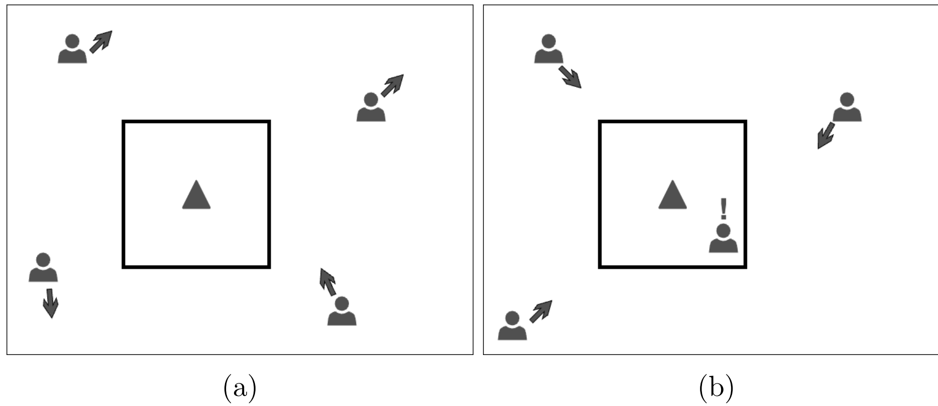



Figure 5.4: 10 birds flock

the individual traces of each bird. Looking at the image, we can deduce that, once every bird is in a stable state, the flock move to the right. In the case where we initialise five birds, we can still understand the behaviour of each one: they start moving right, but suddenly, the mean point of the flock shifts to the left side of the map. The last image, which considers ten birds, does not permit us to appreciate the movements, but it is useful to understand that if they don't reach a stable state, they can not move from a certain position.

## 5.2 FinderBots

In this scenario, a group of robots has to find a target located inside the area. Initially, the finders move randomly, looking for the objective, which can be an injured person (if we imagine the robots working inside a collapsed building) [27] or an infesting grass (if we consider an agricultural scenario) [28]. As we can see in Figure 5.5a, a group of robots roam in an environment looking for the object (the grey triangle). While they roam, the robots may eventually get closer to the target. Once they get close enough to the target, the agent stops moving. However, the system aims to make every agent reach the target zone. Using the previous behaviour may take an indefinite time to complete the task. To avoid this, we use trigonometric formulas to calculate the approximate position of the target zone using the average position of those agents that have reached the goal. Indeed, in Figure 5.5b, an agent has reached the goal area and its position is communicated to the other, which will steer in the approximate direction of the first one.




(a) (b)

Figure 5.5: Finder Bots before (a) and after (b) finding the target zone

### 5.2.1 FinderBots Specification

In Listing 5.4, we can identify three main components: the element *Target*, the agent *Finder* and the *environment*. The `Target` specification is similar to the one we saw before, with two coordinates, both expressed as real values, to locate its position in the environment.

The `Finder` specification considers, along with the direction (`dirx` and `diry`), a state variable `found` that allows the robot to know if it has found a target. As external features, the agent shows its position (`x` and `y`) and a visible `light` that alerts the others if the target has been found. The Finder can observe if it is close to the objective with `target_sensor`, the approximate `angle` of the objective area, and the `toSensor` variable for indicating if the agent is moving towards the target. The agent can perform three different actions: `stop`, `roam`, or `moveto`. When the `stop` action is performed, both directions of the agent are set to `0.0` and the `found` state variable is set to `true`. In contrast, for the roaming action, we give both a random value between `-1.0` and `1.0`. The last action is calculated on the `angle` observation using the cosine (for `dirx`) and sine (for `diry`) functions. Finally, the behaviour states three conditions: 1) if the agent has found the target, it will stop its execution, 2) if the angle is different from zero, the robot will proceed in the target's approximate direction, or 3) otherwise will roam.

The environment is used to update the observations and the features of the Finder robot. In the sensing function, the `target_sensor` observation field is updated, calculating if the agent is in the `tolerance` distance from the target. The `angle` and `toTarget` observations are calculated on the existence of at least one Finder with the `light` feature set to true. If this condition is met, the arctangent function is applied to the mean position of the agents while the other observation flag is activated. Otherwise, they are set to the default value, and thus, the moving towards behaviour is not activated. The

dynamic component is used to update the position of each agent by adding `dirx` and `diry` to the corresponding position feature values and activate the light flag.

```
param na = 3; param tolerance = 2.0;
param tposX = 5.0; param tposZ = 5.0;

agent Finder =
  state:
    real dirx = 0.0;
    real dirz = 0.0;
    bool found = false;
  features:
    real x = 0.0;
    real z = 0.0;
    bool light = false;
  observations:
    bool target_sensor = false;
    real angle = 0.0;
    bool toTarget = false;
  actions:
    stop [ dirx <- 0.0; dirz <- 0.0; found <- true; ]
    roam [ dirx <- U[-1.0 , 1.0]; dirz <- U[-1.0 , 1.0]; found <- false; ]
    moveto [ dirx <- cos(angle); dirz <- sin(angle); found <- false; ]
  behaviour:
    when target_sensor -> [ stop:1; ]
    orwhen toTarget -> [ moveto:1; ]
    otherwise [ roam:1; ]
end

element Target =
  real posx = 0.0; real posz = 0.0;
end

environment :
  sensing:
    Finder [
      target_sensor <- (any Target : distance(posx, posz, it.x, it.z) <
    tolerance);
      if (any Finder: light) then
        let meanX = mean Finder [light] .x
        and meanZ = mean Finder [light] .z
        in
          angle <- angleOf(meanX, meanZ, it.x, it.z);
          toTarget <- true;
        endlet
      else
        angle <- 0.0;
          toTarget <- false;
        endif
    ]
  dynamic:
    Finder [
      x <- x + dirx;
      z <- z + dirz;
      light <- found;
    ]
end
```

Listing 5.4: FinderBot Scenario Specification Snippet

### 5.2.2 FinderBots Simulation

The FinderBot scenario is configured using the parameters in Listing 5.5. In this component of the specification, we need to declare where we will place the objective and where the agents will start their search. For this instance, we choose a static initial position of the target that can be freely changed before simulating the scenario using the command to modify parameters. Instead, the Finder bots are generated inside the same row on different columns. The value of a column (`ox`) is produced by a random real value between 0 and 10. This procedure is repeated `na` times, which has been declared as a parameter at the beginning of the specification.

```
configuration Main :
    for ox sampled distinct na time from U[0, 10] do
        Finder [ x = ox; z = 0.0; ]
    endfor
    Target [ posx = tposX; posz = tposZ; ]
end
```

Listing 5.5: FinderBot Scenario Initial Configuration

Here, we consider a case where the target is located at (7,7) and expect the agents to take some time to find the objective. Figure 5.6 is a possible simulation that may happen with our scenario. Each agent is represented with a different colour, while the static target is shown as a bold cross.

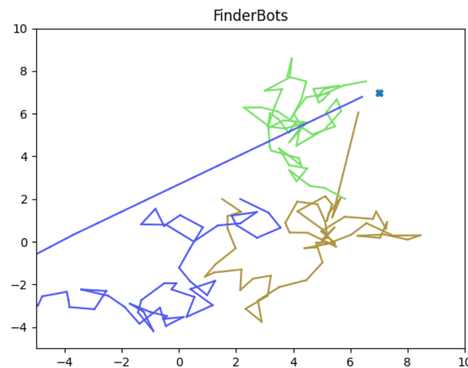



Figure 5.6: Simulation Results of FinderBot

Analysing the figure, we can deduce that the green agent is the first to reach the tolerance zone, signalling to the others the approximate location of the objective. Until this point, the other two had roamed the map without success (for example, the blue one moved in a different direction from the target). Now, they receive the signal from the green agent and can move

towards it. The second successful agent is the brown one, reaching the objective zone. When this happens, the blue agent slightly steers towards the middle point between the actual positions of the green and brown agents.

## 5.3 SEIR

Multi-Agent Systems are widely used in epidemic scenarios to simulate illness transmission. These are used to understand how fast a population gets infected with a disease and, in some cases, how people can contrast the spread of the illness. For example, epidemic models have been widely used in the last COVID-19 global pandemic [115], especially the SEIR [29] and its derived models. In the SEIR model, a group of *susceptible* individuals may come into contact with an infected one. The latter can infect the others, turning them into *exposed*. Exposed people have a determined incubation period that will make them *infected*. Finally, they will *recover* from the illness after some time. In some cases, the model also considers the possibility of becoming infected again. The full cycle is shown in Figure 5.7.

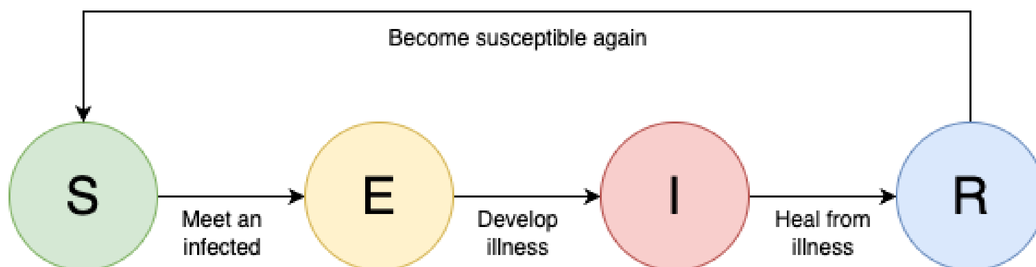



Figure 5.7: The SEIR model cycle

In the proposed version of the epidemic, we consider a continuous evolution of the infection in a person. After being exposed to the virus, the agent moves into an infectious state and then recovers from it. However, the recovered agent could be susceptible again, allowing the cycle to start again.

### 5.3.1 SEIR Specification

The model specified in Listing 5.6 shows how to express a SEIR-based scenario. Because the interactions are only between agents, we need only to specify the `Agent` as the main actor of the scenario. The model considers a group of constants associated with an integer value. These are used to define the actual health of the agent: 0 if *susceptible*, 1 if *exposed*, 2 if infected, and 3 if *recovered*. We can include the initial number of susceptible agents `nSus` and infected ones `nInf` as parameters. We can also declare the exposition

```
const S = 0; const E = 1; const I = 2; const R = 3;

param nSus = 10; param nInf = 3;
param er = 0.15; param ir = 0.25; param rr = 0.25; param lr = 0.1;
param tolerance = 10.0; param speed = 1;

agent Agent =
  state:
    int status = 0;
    real dir = 0.0;
  features:
    bool contagious = false;
    real x = 0.0;
    real z = 0.0;
  observations:
    bool contact = false;
  actions:
    move [dir <- U[ 0.0 , (2 * PI)];]
    getExposed [dir <- U[ 0.0 , (2 * PI)]; status <- E;]
    getInfected [dir <- U[ 0.0 , (2 * PI)]; status <- I;]
    recover [dir <- U[ 0.0 , (2 * PI)]; status <- R;]
    lost [dir <- U[ 0.0 , (2 * PI)]; status <- S;]
  behaviour:
    when contact && (status == S) -> [ getExposed : er; move : (1-er); ]
    orwhen (status == E) -> [ getInfected : ir; move : (1-ir);]
    orwhen (status == I) -> [ recover : rr; move : (1-rr);]
    orwhen (status == R) -> [ lost: lr; move: (1-lr); ]
    otherwise [ move:1; ]
end

environment :
  sensing:
    Agent [
      contact <- (any Agent : (contagious && (distance(it.x, it.z, x, z) <
    tolerance)));
    ]
  dynamic:
    Agent [
      contagious <- (status == I);
      x <- (x + cos(dir)) * speed;
      z <- (z + sin(dir)) * speed;
    ]
end
```

Listing 5.6: SEIR Scenario Specification Snippet

rate `er`, the infection rate `ir`, the recover rate `rr`, and finally the lost rate `lr`. As rates, these four variables should consider only real values between 0 and 1. As additional parameters, we can consider the `tolerance`, as the safety distance between agents, and the `speed`.

The `Agent` state includes two variables. The integer variable `status` refers to the actual health state of the agent, while the `dir` variable is used to express the direction. As external features, agents can show if they are `contagious` or not, and their position (in terms of coordinates on a plane). Concerning the available actions, an agent can perform the basic movement with `move`, or combine it with an action related to the infection. For example `getExposed` assigns to the `status` the code `E` (1), while `getInfected` sets the `status` to `I` (2). Similarly, `recover` and `lost` allow the agent to set the `status` to `R` and `S` respectively. These actions are chosen according to the behaviour. As the first rule, an agent can get exposed or keep moving according to the exposition rate `er`, but only if it has been in `contact` with an infected and its health status is equal to `S`. The other behavioural conditions consider only the `status` and the corresponding action is executed on the related rate.

As usual, the environment manages the sensing function and the dynamics. Concerning the observations, we need only to update the `contact` field. This is set to true if there is at least an `Agent` that is `contagious` and, at the same time, is closer than the `tolerance` distance. On the other hand, the dynamic update function sets the `contagious` field to true if the agent has been `infected`. The coordinates `x` and `z` are modified according to the `dir` and the `speed`.

### 5.3.2 SEIR Simulation

The `Main` configuration in Listing 5.7 considers two types of agents' initialisation. In the first case, we use the parameter `nSus` to create a group of susceptible agents. On the other hand, the second cycle is used to initialise `nInf` infected agents, with the `status` state set to `I` and the `contagious` feature to `true`.

Concerning the trace specification shown in Listing 5.8, we use the `dir` state to show the direction and a triangle as a shape. On the contrary, the colour has been customized to change to red if the agent has been infected, yellow if it has been exposed, blue if it has recovered, and green if it is susceptible.

For this scenario, we are interested in understanding the trends of the agents' state. We are particularly interested in showing how the four populations (infected, exposed, recovered, and susceptible) evolve over time. The plotted results of the simulation, which considers a deadline of 100 time units, can be seen in Figure 5.8. We initialise 90 susceptible agents and 10 infected

```
configuration Main :
  for i sampled distinct nSus time from U[0, (nInf+nSus)/2] do
    Agent [ x = i; z = U[0,50]; ]
  endfor
  for j sampled distinct nInf time from U[0, (nInf+nSus)/2] do
    Agent [ x = j; z = U[0,50]; status=I; contagious = true;]
  endfor
end
```

Listing 5.7: SEIR Scenario Configuration

```
direction = dir;
shape = triangle;
colour = {
    when (status == 2) : red;
    when (status == 1) : yellow;
    when (status == 3) : blue;
    otherwise green;
}
```

Listing 5.8: Tracing Specification for SEIR

agents for this example. To change state, the agents use the same rates presented in the specification, which means that `er` is equal to 0.15, `ir` to 0.25, `rr` to 0.25, and `lr` to 0.1.

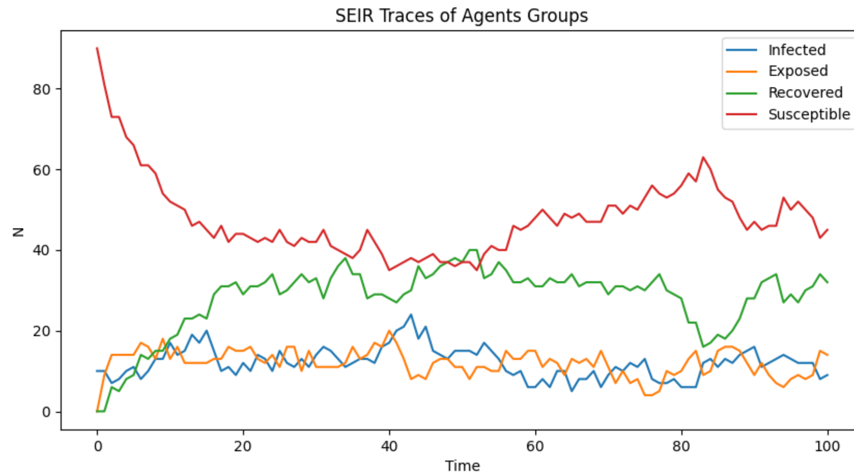



Figure 5.8: Simulation Results of SEIR

The simulated infection shows that many susceptible agents become exposed to the disease due to the mild exposition rate. As expected, the number of infected and contagious agents increases right after. The time spent by each agent in an illness state (exposed or infected) is quite low, also because there is a 25% of chances to recover from the illness after one step. The

number of recovered agents reaches its peak at half the simulation, while after 30 time units, the recovered agents become susceptible again. In this model, safety distance also plays a crucial role. Indeed, this first simulation considers a `tolerance` of 10 space units.

However, if we set it at 7.5, which means that the agents should be closer to each other to get exposed, we can see that there is only a pit of susceptible agents around the 20th step, but then the whole, healthy population increases again. This case is shown in Figure 5.9.

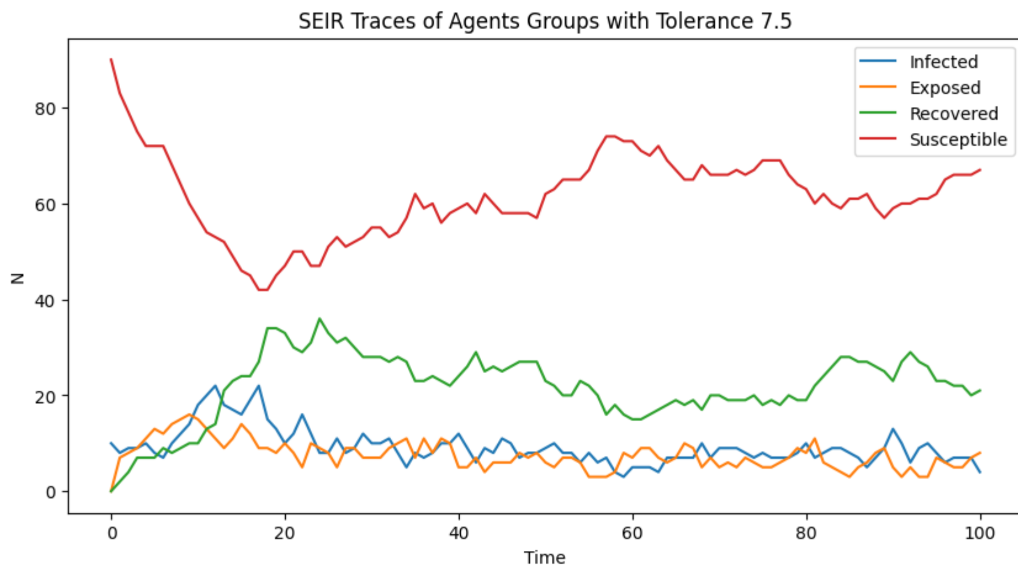



Figure 5.9: Simulation Results of SEIR with 7.5 `tolerance`

The proposed plots consider only one simulation of our SEIR model. We recall that Sibilla is equipped with a set of functionalities to study trendings of multiple simulations. In Figure 5.10, we can analyse the mean trendings of a group of measures: the number of agents belonging to a particular health status. Indeed, we can deduce that Figure 5.8 is compliant to this last plot.

## 5.4 Red-Blue

Agent-based systems can show how consensus dynamics work. Consensus refers to a group of actions allowing the agents to agree on certain values or states [30]. Many protocols have been formalised in the literature to mimic politics and other behavioural scenarios. Indeed, some authors use binary consensus to simulate decisional processes [116].

In our consensus scenario, we consider a population of agents that can be members of the *Red* party or of the *Blue* one. Figure 5.11 shows how

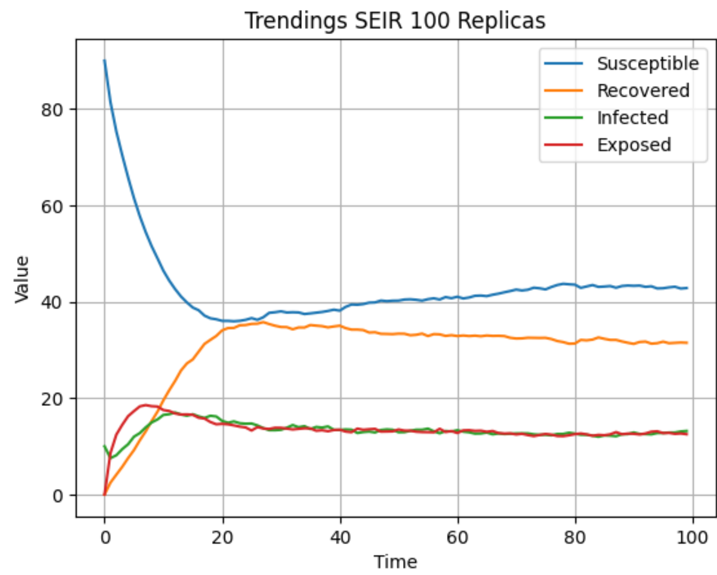



Figure 5.10: 100 Replicas of SEIR Model

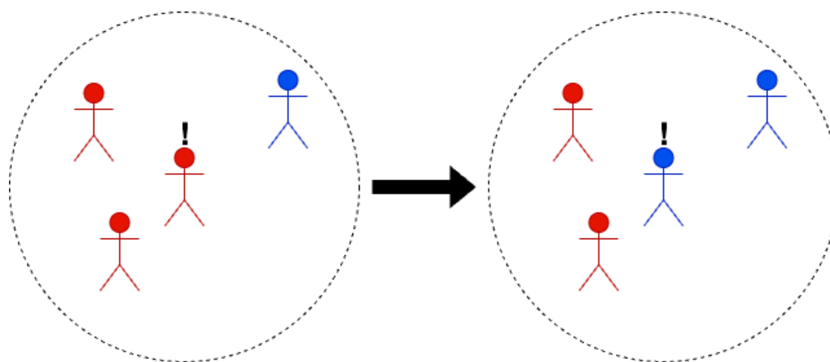



Figure 5.11: Red-Blue Scenario Interaction

an agent evolves. After observing the others close to it, the concerned agent perceives how many agents are in its group and then decides to keep the same party or switch. In the image, the central red agent sees that it finds itself surrounded by two other red agents and a blue one. In the next iteration, the agent becomes blue to support the opposite faction.

### 5.4.1 Red-Blue Specification

The full specification of the Red-Blue scenario is described in Listing 5.9, which considers the `Supporter` as the main interacting entity. The model considers the following parameters: to assign the agents to the two parties, we use respectively `nRed` and `nBlue`, while `tolerance` is the considered distance by every agent, `speed` how fast it can move, and `cr` is the rate for changing the party.

The `Supporter` knows its `direction`, described as a real value, and if it has decided to change its colour (with the boolean attribute `switchParty` ). As external features, the agent shows its position coordinates and is a member of the `blueParty` or the `redParty`. Lastly, observation attributes are composed of a boolean that signals the agent if it has been `challenged` by the other surrounding agents and a real variable `delta`, which is based on the number of the surrounding entities. The `Supporter` can only perform two tasks: the basic movement, allowing it to choose a direction and keep the same colour status, and the `changeParty` action, which considers a new direction and sets the `switchParty` variable to `true`. Finally, the behaviour is simply based on the `challenged` observation variable. In a positive case, the agent may decide to perform the `changeParty` action with a probability of `delta*cr` or perform the basic `move` action.

Concerning the environment, it updates the observation `challenged` by considering the number of red and blue agents around the observer and its colour. The condition is activated when the considered agent is surrounded by more agents of the same party than the other. In this case, the `Supporter` is `challenged`, and the variable is activated. The `delta` observations field is updated according to the absolute value of the difference between the number of surrounding red agents and the blue ones, divided by the sum of these two. On the other hand, the dynamic function first updates the position based on the direction vector and then the corresponding colour using an XOR mechanism: the attribute is set to `true` only if the agent colour feature is active or the agent has decided to switch. If both conditions are met, the feature is deactivated.

```
param nRed = 70;
param nBlue = 30;
param tolerance = 10;
param speed = 1;
param cr = 1;

agent Supporter =
  state :
    real direction = 0.0;
    bool switchParty = false;
  features :
    bool redParty = false;
    bool blueParty = false;
    real x = 0.0;
    real z = 0.0;
  observations :
    bool challenged = false;
    real delta = 0.0;
  actions :
    changeParty [direction <- U[0.0, 2*PI]; switchParty <- true;]
    move [direction <- U[0.0, 2*PI]; switchParty <- false;]
  behaviour :
    when challenged -> [changeParty : delta*cr ; move : (1-delta*cr);]
    otherwise [move:1;]
end

environment :
  sensing :
    Supporter[
      let actualBlue = # Supporter[blueParty && (distance(x,z,it.x,it.z)<
    tolerance)]
      and actualRed = # Supporter[redParty && (distance(x,z,it.x,it.z)<
    tolerance)]
      in
        if (((actualRed+actualBlue)>0)&&((it.redParty && (actualBlue<
    actualRed))||(it.blueParty && (actualBlue>actualRed)))) then
          challenged <- true;
          delta <- abs(actualBlue-actualRed)/(actualRed+actualBlue);
        else
          challenged <- false;
          delta <- 0.0;
        endif
      endlet
    ]
  dynamic :
    Supporter[
      x <- (x + cos(direction)) * speed;
      z <- (z + sin(direction)) * speed;
      redParty <-(redParty && !switchParty)||(!redParty && switchParty);
      blueParty <-(blueParty && !switchParty)||(!blueParty && switchParty);
    ]
end
```

Listing 5.9: Red-Blue Scenario Specification

### 5.4.2 Red-Blue Simulation

This simulation is based on a configuration (Listing 5.10) considering 100 units: `nRed` is equal to 70, while `nBlue` to 30. The first group will be assigned to the `redParty`, while the other is assigned to the `blueParty`. The entities are positioned, at the beginning, at a random point inside a 100x100 square space.

Listing 5.11 shows the trace specification for the Red-Blue scenario. In this example, `direction` is omitted because the specification recognises the same attribute from the agent declaration. We chose the triangle as the main shape, and the assigned colour is associated with the corresponding party.

```
configuration Main :
  for i sampled distinct nRed time from U[0,100] do
    Supporter[redParty = true; blueParty = false; x = i; z = U[0,100];]
  endfor
  for j sampled distinct nBlue time from U[0,100] do
    Supporter[redParty = false; blueParty = true; x = j; z = U[0,100];]
  endfor
end
```

Listing 5.10: Red-Blue Scenario Configuration

```
shape =  triangle;
colour = {
    when redParty : red;
    when blueParty : blue;
    otherwise gray;
}
```

Listing 5.11: Tracing Specification for Red-Blue

The model's objective is to see if the agents reach a consensus among them. We are interested in having both populations around 50% The plotted result in Figure 5.12 considers 100 time units and the changing rate `cr` is set to 1, thus it does not afflict `delta`.

The trends are symmetrical, as we can see, because the agent can only be blue or red. The goal is immediately reached in the plot, even if the agents change their colour repeatedly. This is caused by the slight concentrations of entities around an agent, which makes it easier for it to make the change. Indeed, the case presented in Figure 5.13 shows how more agents, in this case, 300 blue agents and 700 red ones, can mitigate the effect seen before.

Similarly to the SEIR scenario, we want to know the average trends of the Red-Blue scenario. In Figure 5.14, we analyse how agents behave on average. The plot considers 100 simulation replicas and the original configuration of 70/30 agents. Unlike the single trace plot, the population trend is smoother. However, the larger sections mirror what happens in the single

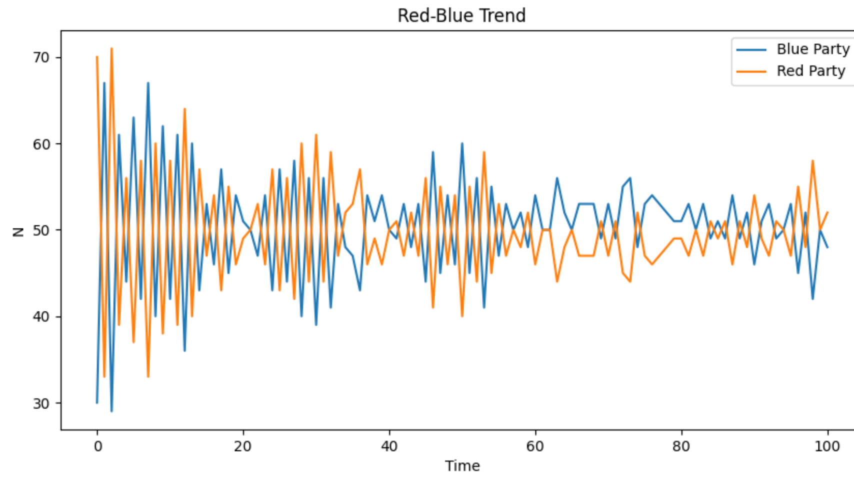



Figure 5.12: Simulation Results of Red-Blue

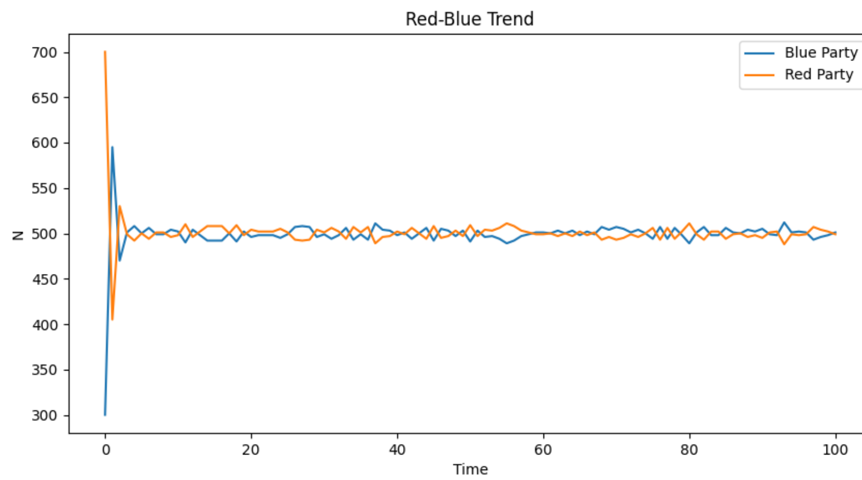



Figure 5.13: Simulation Results of Red-Blue with 1000 agents

trace (for example, the expansion after the 40th step is similar to the one in Figure 5.12).

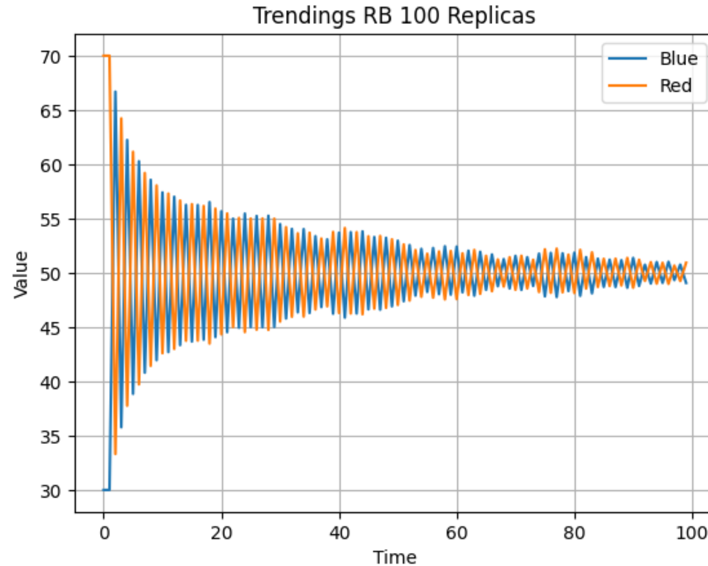



Figure 5.14: 100 Replicas of RB Model

## 5.5 Concluding Remarks

In this Chapter, we have seen how different MAS models can be described in YODA. Each of the considered scenarios allowed us to highlight the role of the mechanisms provided by YODA. Moreover, thanks to the simulation features provided by Sibilla, we have obtained a view of the possible behaviours of the considered models. With the Flock, which will act as the new running scenario, we showed how we can express coordination among agents. The plotted results of the birds' movements demonstrated how the flock remained cohesive during the simulation. In the FinderBot example, we appreciated how each agent can drastically change their behaviour after an event. Indeed, agents that are further from the objective turn towards the target once they receive the signal. Finally, the classic SEIR model and the Red-Blue example give insight into how agents interact and modify their state according to their perception of their surroundings.

# CHAPTER 6

# VISUALISING YODA SIMULATIONS

*Through the Force,*
*things you will see.*
*Other places.*
*The future... the past.*

Simulations of CASs generate a determined quantity of possible computations. Being able to represent results well is crucial, and it permits those who are not used to reading data to have a better insight into what a system may produce. The authors of [117] stress the attention on *how* simulations should be represented, highlighting the fact that poor visualisation may be counterproductive. To strengthen the importance of visualisation, the authors of [118] individuate different issues during the process, which starts from the data collection and ends with the representation of knowledge.

Visualisation methods can range from more static ones to more complex replays. This distinction depends on the level of detail we want to achieve. On the one hand, analysts can perform statistical analysis from the results of graphs, charts, or diagrams. This is useful when working on quantitative analyses which require the study of trends and average evolution of systems [119]. In the other case, visualisation is based on graphical solutions. These are used in various application domains, like robotics [120] or ecology [121], and usually exploit graphics processing power rather than the computer one. Plotting a graph is an easier task than creating a real-time visualisation tool. Many libraries already exist, starting from the well-known Python Matplotlib [122], and they can support different types of analysis.

With this chapter, we introduce Sequit[1], a tool that, integrated with the Sibilla simulator, permits rendering in 2D and 3D environments the evolution of a set of agents [31]. The visualiser should overcome the necessity of replaying agent traces, allowing us to appreciate even more how agents move and act. Indeed, other approaches are often too specialised by referring to specific languages or frameworks, as shown in Section 2.3. On the contrary, Sequit aims to provide an agnostic solution: by following the same philosophy of Sibilla, it does not rely on a single language.

Moreover, by relying on Unity technology, we can build Sequit and run it on many existing architectures, allowing designers to integrate the tool as an external component in the simulation pipeline.

## 6.1 Implementing Sequit

While Sibilla has been developed in Java and takes advantage of computational efficiency, from the graphical point of view, we need a more complete and well-structured solution that can host simulation visualisation. For this reason, we rely on the well-known Game Engine *Unity*[2], which is a real-time environment used not only in gaming, but also in movie making, architecture, and automotive. The term *Real-time* describes images' rendering frequency and the possibility of dynamically interacting with the application. As a Game Engine, Unity serves as a convergence point of different developing aspects, allowing assets to interact. In this section, we discuss how the module is implemented. We will see how the classes and available game objects work so that we can replay each agent's simulation traces.

### 6.1.1 Architecture

In Unity, classes and objects are included in what are technically called *Game Objects* (GO). These are the foundations for scenes (the environment where actions take place) and "*act as a container for functional components which determine how the Game Object looks and what the Game Object does.*" [3] A GO can be anything belonging to a single scene, from a 3D model to a camera or even the user interface. Our solution has been developed in a single scene including the necessary GOs. They are represented as three invisible GOs allowing the management of the simulation. These are: *Button Manager*, *Game Manager*, and *Simulation Manager*. Moreover, we included a set of utility classes to read files. Figure 6.1 shows an overview of the architecture

[1]The tool is available at `https://github.com/quasylab/Sequit`
[2]Full site at `https://unity.com/`
[3]`https://docs.unity3d.com/Manual/class-GameObject.html`

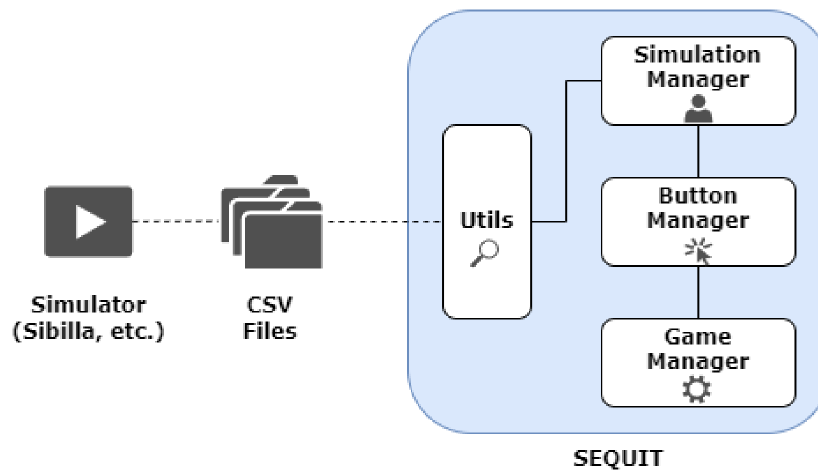



Figure 6.1: Sequit Architecture

and how each component interacts with each other or external resources, while Figure 6.2 represents the actual GOs.

Figure 6.2: Scene Hierarchy of Sequit

**Button Manager.** This object manages how the user can directly interact with the tool using the available buttons. The assigned script contains the methods invoked when one of the corresponding buttons is pressed. For example, if we press the *Quit* button, the GO calls the `QuitApplication` method and closes the app; if we click on the *Camera* button, the `ChangeCamera` method changes the camera type. Due to its nature, the Button Manager interacts with the Game Manager, the Simulation Manager, and the Camera object.

**Game Manager.** The Game Manager is a common object used while developing applications within an engine. Usually, it manages and coordinates

the interactions between the user and the game, e.g., by showing the collected points or the player's remaining life. In Sequit, the Game Manager is conceived to navigate between the interfaces and control eventual keyboard inputs, which are used to move around the visualisation, close the action menu, and more. However, its main task is to modify the time scale of the visualisation. The attached script contains a set of methods, that exploit Unity's libraries, to stop, resume, and even change the velocity of the simulation.

**Simulation Manager.** The Simulation Manager GO is responsible for instantiating and managing the acting agents of the visualisation. Depending on the type of agent we want to visualise, the manager generates 2D or 3D agents based on an interface called `IAgentController`. In both cases, the implemented controllers should be able to change shape and colour and proceed to the next step.

**Utils.** The Utils namespace contains the utility classes for reading and interpreting the .csv files. Two custom interfaces have been developed (`IFileReader` and `IStepsController`) from which we derive two classes: the `CsvFileReader` class and the `FileStepsController` class. The file reader is responsible for interpreting the table and is based on the .NET native class `StreamReader`. Apart from the constructor, we use the existing library to read the lines. The other utility class checks the correctness of each line parsed by the reader and scrolls up and down the data table. A single line in the given file is represented by the class `Step` It consists of five variables providing the information the agent controller can use to move or visually inform of a state change. These are:

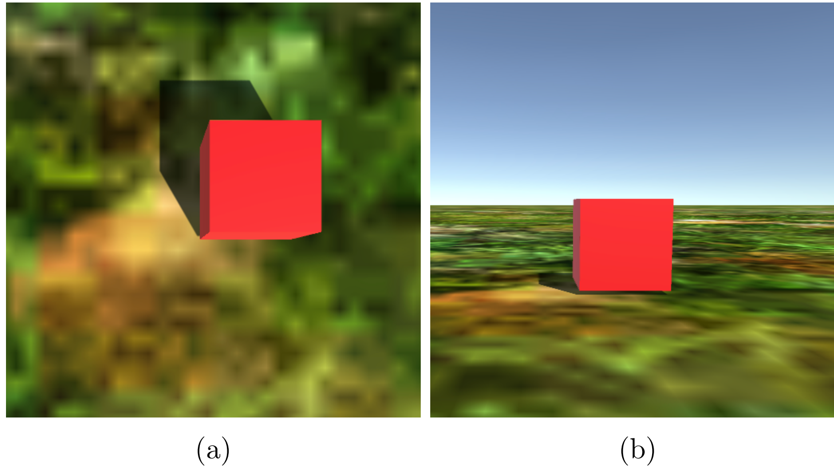
- **Time**, a float variable to measure the time steps;
- **Position**, a Vector3 variable used to move the agent in the space;
- **Rotation**, a Vector3 variable used to rotate the agent in a certain direction;
- **Shape**, a string variable used to change the shape (sprite or model);
- **Color**, a string variable used to change the colour of the agent.

It should be noted that the previous format must be the one used in the .csv file. This file format can be obtained from the Sibilla trace command or other sources and software.

### 6.1.2 Features

Sequit has been implemented keeping in mind the basic needs of a common user. As a lightweight tool, its objective is to provide a solution that can be used immediately. The actual version of the visualiser has the following functionalities: *two degrees of freedom* for the camera, *controlling* the simulation, and adding *2D* and *3D agents*. In this section, we discuss said features.

**Switching Cameras.** Our solution permits the user to visualise the replay of traces from two points of view. By clicking on the corresponding button, the user can switch between the two camera modes, shown in Figure 6.3: *eagle-eye* and *rotatory*. In the first case (Figure 6.3a), the camera is put above the plane where the agents operate and allows the user to better comprehend 2D cases or those where agents move in two directions. With this kind of camera, the user can navigate with WASD/Arrow keys and move away with the mouse wheel to get a better view of the scene or get closer to appreciate the details. The rotatory camera movement (Figure 6.3b) is based on focusing on a focal point located in the middle of the scene and can be useful for inspecting 3D scenarios. It uses the same keybindings to move around in a spherical way.



(a) (b)

Figure 6.3: The two different available cameras

**Simulation Control.** The agents move in the scene according to the provided traces. However, one may be interested in understanding how the entities are located in the environment at a certain moment. For this reason, the visualiser implements a set of methods that, using basic Unity constructs, permits the management of the velocity of the visualised agents. By pressing the corresponding button (or the spacebar key), we can stop (and resume)

the time, allowing us to inspect the situation. It is also possible to speed up or slow down the visualisation. The actual velocity of the simulation and the utility buttons are displayed in the user interface, as shown in Figure 6.4.

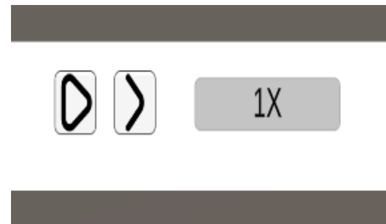



Figure 6.4: UI of the visualisation speed

**Generating Agents.** The fulcrum of the visualiser is the possibility of loading the traces and generating the agents. Sequit provides a form where it is possible to configure the visualisation as the user desires. Initially, we must decide whether we want to generate 2D agents (as sprites) or 3D agents (as models). Independently of the type, we can specify the directory path containing the .csv files in the text field. The final step is to choose an optional preloaded plain among those available. Once we press the generation button, the visualiser reads the provided data and generates an entity for each available file. The agents will start moving right after the data has been loaded correctly.

## 6.2 Visualising the Flock Scenario

In this section, we will show how the flock example, introduced in Chapter 5, can be visualised using Sequit. In particular, we will show how a given set of traces can be visualised in Sequit[4]. We can remark that, if needed, a *model specific* visualisation environment can be developed by using the Unity Editor. However, in this section, we will focus only on the use of Sequit as a *stand-alone* tool.

Once we are provided with the simulation traces from our model, we can load them into Sequit. When Sequit is executed a simple user interface is provided to start the visualisation or to quit the application (see Figure 6.5). By pressing the button *Start*, the dialogue window of Figure 6.6 is provided. This can be used to select the kind of model to render (2D or 3D), the directory where the simulation traces are located, and the *plane* to use as a background.

[4] A demo video is available at http://quasylab.unicam.it/sibilla/sequit/

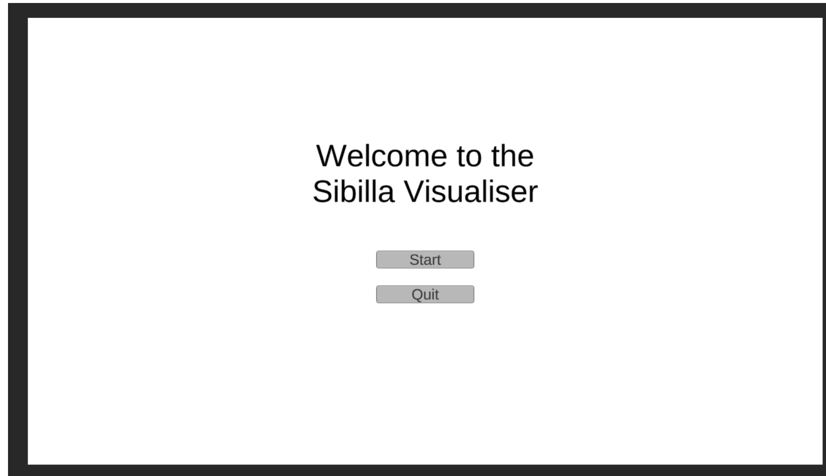



Figure 6.5: The starting screen of Sequit

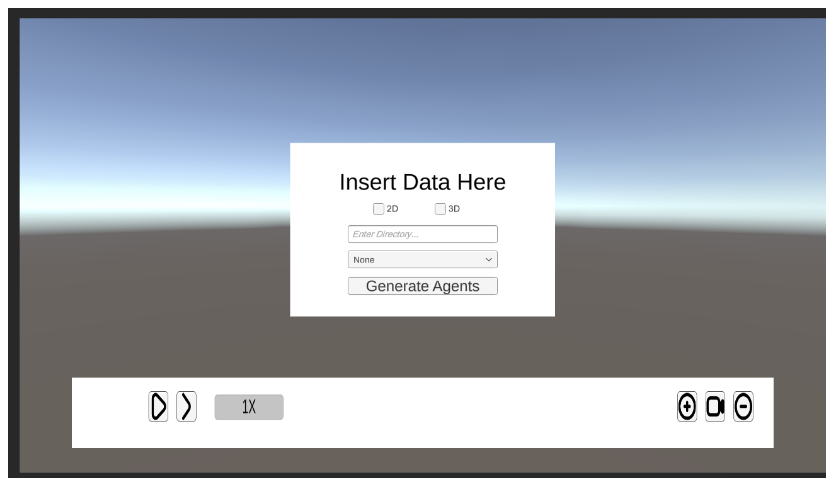



Figure 6.6: The initial interface with the agent form

When all the needed parameters are provided, the visualisation can be started by pressing the button *Generate Agents*. The tool instantiates all the agents that will start moving according to the info saved on the loaded traces. The button bar placed at the bottom of the window can be used to control the visualisation. On the left, we can find the buttons that can be used to *start/pause* the video and to control the *playback speed*, which can be used to slow down or speed up the video. On the other side, the buttons that control the camera, as described in Section 6.1.2, are placed. In our example, we have used the 2D configuration, the "Sky" plane, and the eagle-eye view camera. We can observe that the birds appear in a different colour depending on the distance from the mean point of the flock: blue if it is too far, red if too close, and yellow if it is at the right distance range. In Figure 6.7 we can see the five birds and, as expected, the middle one is red, because it is too close to the center, the two yellow ones are at the right distance range, and finally, the blue ones are too far from the mean point.

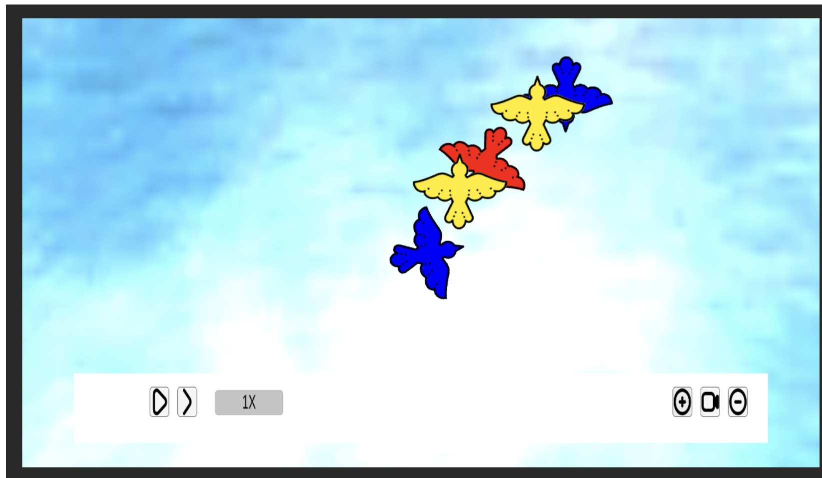



Figure 6.7: The Flock at the beginning of the simulation

In Figure 6.8, we present a screenshot of a time instant where all the birds are far from the mean point. As we can see, here all the birds are coloured blue, which they need to get closer, according to the specified behaviour.

## 6.3 Visualising the Other Scenarios

In Chapter 5, we implemented three additional advanced models in YODA. Let us recall that every model considers spatial features, thus we can replay the traces in Sequit. In this section, we discuss the visualisation of these three scenarios.

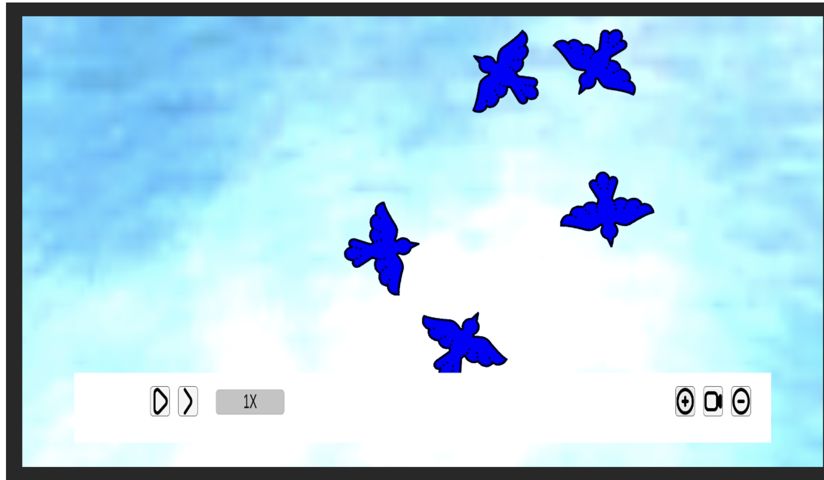



Figure 6.8: The Flock after a while

### 6.3.1 Visualising FinderBot

For this simulation, we instantiate 10 agents looking for the target. They are represented as cubes because we use the `square` shape in the trace specification and a three-dimensional environment. In Figure 6.9, we can see them coloured red because they have not found the target yet. In this phase, they roam around randomly, looking for the target. We preventively know that the objective is located at `[5.0, 5.0]`, but it is not visible in the screenshots because it is not an agent, thus we can not produce a trace.

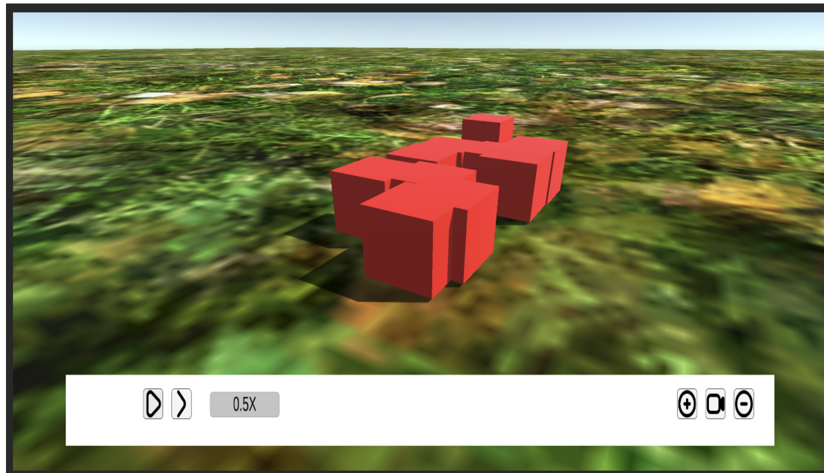



Figure 6.9: FinderBot in Sequit at the beginning

After a few rounds, some agents find the target, becoming green. Figure 6.10 is a screenshot of the moment when this happens, and the information about the target's location is sent to the other agents. The agents

moving to said position are yellow-coloured and, in the simulation, move towards the mean position of the two green finders.

Figure 6.10: FinderBot in Sequit after finding the target

### 6.3.2 Visualising SEIR

The SEIR model simulates an epidemic scenario: an agent can be exposed to a virus at a certain rate, transition to an exposed state, and finally become ill. The simulation uses a pool of 100 agents, of which 90 are susceptible, and 10 are infected. They are randomly instantiated in a square area of 50x50. As shown in Figure 6.11, the three populations (healthy, exposed, and infected) are fairly distributed in the area. Considering the model proposed in Listing 5.6, we can expect the second group, represented as yellow triangles, to be the most prominent. In fact, the exposition rate is higher than the other two. On the other hand, red agents (infected) and green agents (healthy) are nearly balanced in the image.

### 6.3.3 Visualising Red-Blue

In the Red-Blue case study, we want to understand how two species of agents react when surrounded by members of the same and opposite party. In order to change parties, an agent should see a bigger group of the same side than of the opposite one. The goal of the model is to have a fairly balanced population of agents belonging to both parties. Figure 6.12 shows a shot where we reached the objective. The played simulation considers 70 red agents and 30 blue ones. From the specifications seen before, we know that the agents reach the goal quickly.

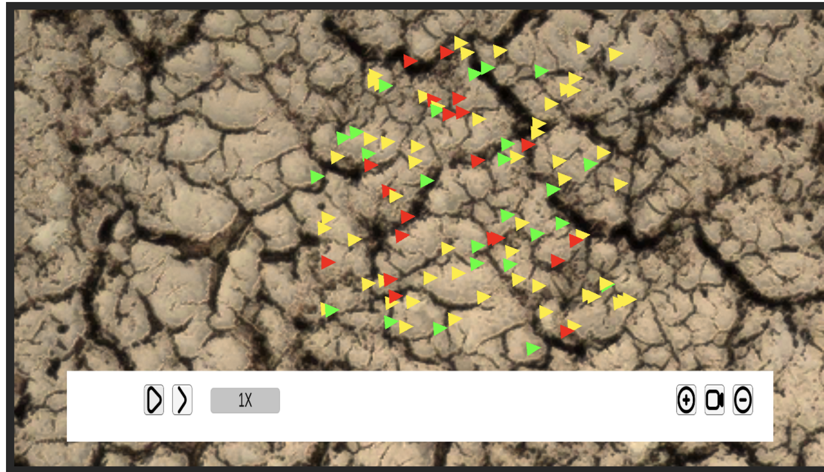



Figure 6.11: SEIR Model in Sequit

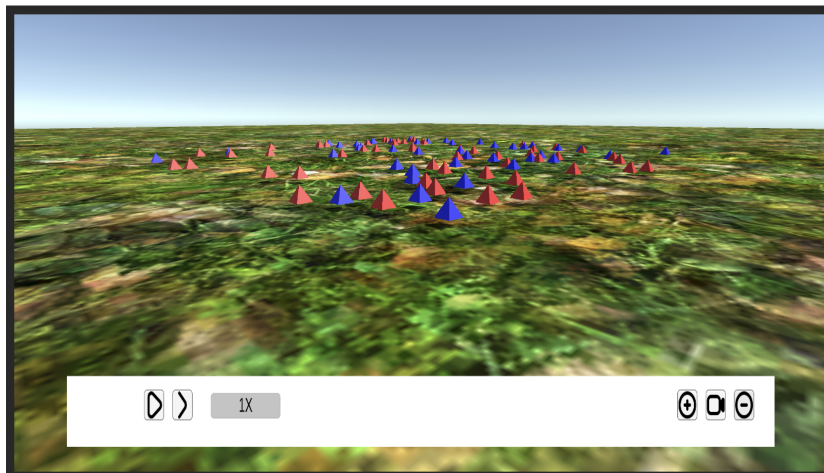



Figure 6.12: Red-Blue Model in Sequit

## 6.4 Future Developments of Sequit

For this part of the chapter, we would like to highlight some future developments for Sequit. These are discussed here because they are strictly related to the tool and not to the whole project. We plan to introduce more functionalities designed to improve the user experience. For example, we would like to add a third degree of freedom for the camera, able to follow a specific agent during its trace execution: this agent could be selected either via the simulation window (by a click) or via appropriate queries depending on the agent attributes. For what concerns the visualisation control, we would like to implement a controllable progress bar to scroll between time instants of the simulation, along with some shortcut buttons for restarting and rewinding the agents. Currently, the application is built to support Windows, macOS, and Linux distributions, but we would like to provide an additional build supporting WebGL [123].

Two major improvements are planned for improving optimisation and usability. In the first case, we would like to explore and add Unity *DOTS*[5] technology. This is based on the *ECS* (Entity-Component-System) paradigm and it is considered as "*a combination of technologies and packages that delivers a data-oriented design approach to building games in Unity*". This means that data rely on a single function instead of having an object-oriented approach. DOTS could be used to represent those systems requiring many, identical entities, like Population Models. The second major improvement we are planning is the development of a *Unreal Engine*-based version of Sequit. While the actual version is lightweight and does not require much effort from the user, there are some limitations from a flexibility and openness point of view due to the engine's architecture and compiler choice (C# compiled to C++). Instead, Unreal Engine [124] projects are compiled directly in C++, allowing the user to customise the application extensively. An Unreal Engine version could permit a more straightforward process, immediately connecting Sibilla with Sequit.

[5] https://unity.com/dots

# CHAPTER 7

# MONITORING YODA MODELS

*Difficult to see;*

*always in motion is the future.*

In the spirit of verifying models, model-checking algorithms are developed to check if some properties hold true. In the literature, we can find many methods of performing such tasks. The authors of [125] use Probabilistic Computation Tree Logic (PCTL) to check various properties of CASs with an on-the-fly model-checking algorithm. Similarly, the approach seen in [126] considers Continuous Stochastic Logic (CSL), which is based on modelling via single-clock Deterministic Timed Automaton (DTA). While the first approach considers only global formulas, the other deals with local ones. Considering the approach of [127], the authors show an extension of Linear Temporal Logic (LTOL). This framework is used to specify properties of Multi-Agent Systems and verify against Doubly-Labeled Transition Systems (DLTS).

For this thesis, we used GLoTL as a framework to verify YODA models. The *Global and Local Temporal Logic* (*GLoTL*) has been introduced in [128] and is used to verify properties of Collective Adaptive Systems at both *global* and *local* levels. In the introductory paper, the authors use this model-checking language to verify the properties of Multi-Agent Discrete Time Markov Chain (MA-DTMC) models. The algorithm is based on an on-the-fly approach [125], meaning that the algorithm follows a top-down approach without requiring global knowledge of the complete state space. GLoTL follows a more linear time approach and takes inspiration from the

*Signal Temporal Logic* (STL) [129]. The logic is equipped with syntax and semantic constructs to work with MA-DTMCs.

This chapter shows how we can integrate GLoTL logic in YODA. We start by introducing the model-checking framework and discussing the logic formulas. Then, we benefit from the proposed approach to check if the expected properties are satisfied by a YODA model. We use the Flock behaviour seen in the previous chapters as a running example.

## 7.1 GLoTL Syntax and Semantics

We recall both the syntax and semantics of GLoTL presented in [128]. Starting from standard *boolean operators* (true, $\neg\,\cdot$, and $\cdot \wedge \cdot$), and (*bounded*) *linear temporal operators* ($\mathcal{X}\,\cdot$ and $\cdot\,\mathcal{U}^{[k_1,k_2]}\,\cdot$), we consider two sets of formulas (*local* $\Lambda$ and *global* $\Gamma$).

Global Formulas

$$\Phi \quad ::= \quad \mathsf{true} \;\big|\; \neg\Phi \;\big|\; \Phi_1 \wedge \Phi_2 \;\big|\; \%[\phi] \bowtie p \;\big|\; \mathcal{X}\;\Phi \;\big|\; \Phi_1\;\mathcal{U}^{[k_1,k_2]}\;\Phi_2$$

Local Formulas

$$\phi \quad ::= \quad \mathsf{true} \;\big|\; \mu \;\big|\; \neg\phi \;\big|\; \phi_1 \wedge \phi_2 \;\big|\; \mathcal{X}\;\phi \;\big|\; \phi_1\;\mathcal{U}^{[k_1,k_2]}\;\phi_2$$

Figure 7.1: Syntax of GLoTL formulas.

**GLoTL Syntax.** As we can see from Figure 7.1, *Local formula* $\phi$ are used to specify properties of single agents, built from the *atomic predicate* $\mu$. For the sake of simplicity, $\mu$ is defined as a function of the form $\mathsf{Ag}^* \times \mathsf{Ag} \to \mathbb{B}$, where $\mathbb{B} = \{\top, \bot\}$ As an atomic predicate, $\mu$ can be seen as a *boolean expression* involving attributes of *an agent* and comparing it with other agents' attributes. In the case of the flock, we can use an atomic predicate to check if one *bird* is *next to* or not the *centre of mass* of the perceived flock:

$$\mu_{goal}(S, A) = distance\left(A(x), A(y), \frac{\sum_i S[i](x)}{|S|}, \frac{\sum_i S[i](y)}{|S|}\right) < \delta_{goal}$$

where $\delta_{goal}$ is the *expected distance* between an agent and the centre of mass.

*Global formulas* $\Phi$ can be used to specify properties at a global level. The novel operator $\%[\phi] \bowtie p$ is used to specify that, at a certain point in the computation, the *fraction of agents* satisfying *local formula* $\phi$ is $\bowtie p$, where $\bowtie\, \in \{\leqslant, <, >, \geqslant\}$. For example, a property that one can be interested

in verifying is that the fraction of agents that are next to the centre of mass is greater than 0.75. The formula can be expressed as follows:

$$\Phi_{close} = \%[\mu_{goal}] > 0.75 \tag{7.1}$$

Given the above definitions, we can derive other logical operators as additional macros. A list of derivable operators is reported in Figure 7.2 for both *global* and *local* formulas: *disjunction*, $\vee$, and *implication*, $\rightarrow$, from $\wedge$ and $\neg$; *eventually*, $\Diamond^{\leqslant k}$, and *globally*, $\Box^{\leqslant k}$, from $\mathcal{U}^{\leqslant k}$ .

| GLOBAL FORMULAS | | | LOCAL FORMULAS | | |
|---|---|---|---|---|---|
| $\mathsf{false}$ | $=$ | $\neg\mathsf{true}$ | $\mathsf{false}$ | $=$ | $\neg\mathsf{true}$ |
| $\Phi_1 \vee \Phi_2$ | $=$ | $\neg(\neg\Phi_1 \wedge \neg\Phi_2)$ | $\phi_1 \vee \phi_2$ | $=$ | $\neg(\neg\phi_1 \wedge \neg\phi_2)$ |
| $\Phi_1 \rightarrow \Phi_2$ | $=$ | $\neg\Phi_1 \vee \Phi_2$ | $\phi_1 \rightarrow \phi_2$ | $=$ | $\neg\phi_1 \vee \phi_2$ |
| $\Diamond^{[k_1,k_2]}\Phi$ | $=$ | $\mathsf{true}\ \mathcal{U}^{[k_1,k_2]}\ \Phi$ | $\Diamond^{[k_1,k_2]}\phi$ | $=$ | $\mathsf{true}\ \mathcal{U}^{[k_1,k_2]}\ \phi$ |
| $\Box^{[k_1,k_2]}\Phi$ | $=$ | $\neg\Diamond^{[k_1,k_2]}\neg\Phi$ | $\Box^{[k_1,k_2]}\phi$ | $=$ | $\neg\Diamond^{[k_1,k_2]}\neg\phi$ |
| $\%[\phi] \in [a,b]$ | $=$ | $\%[\phi] \geqslant a \wedge \%[\phi] \leqslant b$ | | | |
| $\%[\phi] \notin [a,b]$ | $=$ | $\neg\%[\phi] \in [a,b]$ | | | |

Figure 7.2: Derivable Logical Operators

Given a global formula $\Phi$ (resp. a local formula $\phi$), the *horizon* of $\Phi$ (resp. $\phi$) is the max number of time steps that are needed to guarantee its satisfaction. The function horizon is inductively defined in Figure 7.3 where $\Psi$ denotes either a global or a local formula.

$$\begin{aligned}
\mathsf{horizon}(\mathsf{true}) &= 0 \\
\mathsf{horizon}(\mu) &= 0 \\
\mathsf{horizon}(\%[\phi] \bowtie p) &= \mathsf{horizon}(\phi) \\
\mathsf{horizon}(\neg\Psi) &= \mathsf{horizon}(\Psi) \\
\mathsf{horizon}(\Psi_1 \wedge \Psi_2) &= \max\{\mathsf{horizon}(\Psi_1), \mathsf{horizon}(\Psi_2)\} \\
\mathsf{horizon}(\mathcal{X}\ \Psi) &= 1 + \mathsf{horizon}(\Psi) \\
\mathsf{horizon}(\Psi_1\ \mathcal{U}^{[k_1,k_2]}\ \Psi_2) &= k_2 + \max\{\mathsf{horizon}(\Psi_1) - 1, \mathsf{horizon}(\Psi_2)\}
\end{aligned}$$

Figure 7.3: Horizon of Local and Global Formulas

**GLoTL Semantic.** Against *global* end *local* computations, we can evaluate the satisfaction of *global* and *local formulas.* In Figure 7.4 and Figure 7.5, we define the satisfaction relations $\models$ and $\models_\ell$.

$$
\begin{array}{llll}
\pi & \models & \mathsf{true} & \\
\pi & \models & \neg\Phi & \iff \pi \not\models \Phi \\
\pi & \models & \Phi_1 \wedge \Phi_2 & \iff \pi \models \Phi_1 \wedge \pi \models \Phi_2 \\
\pi & \models & \%[\phi] \bowtie p & \iff \frac{|\{i|\pi\downarrow i \models_\ell \phi\}_{i<\#\pi[0]}|}{\#\pi[0]} \bowtie p \\
\pi & \models & \mathcal{X}\ \Phi & \iff \pi[1..] \models \Phi \\
\pi & \models & \Phi_1\ \mathcal{U}^{[k_1,k_2]}\ \Phi_2 & \iff \exists h\ k_1 \leqslant h \leqslant k_2.\ \pi[h..] \models \Phi_2 \\
 & & & \qquad \wedge \forall 0 \leqslant i < h.\ \pi[i..] \models \Phi_1
\end{array}
$$

Figure 7.4: Global Formulas: Satisfaction relation

$$
\begin{array}{llll}
\pi_\ell & \models_\ell & \mathsf{true} & \\
\pi_\ell & \models_\ell & \mu & \iff \mu(\pi_\ell[0]) = \top \\
\pi_\ell & \models_\ell & \neg\phi & \iff \pi_\ell \not\models_\ell \phi \\
\pi_\ell & \models_\ell & \phi_1 \wedge \phi_2 & \iff \pi_\ell \models_\ell \phi_1 \wedge \pi_\ell \models_\ell \phi_2 \\
\pi_\ell & \models_\ell & \mathcal{X}\ \phi & \iff \pi_\ell[1..] \models_\ell \phi \\
\pi_\ell & \models_\ell & \phi_1\ \mathcal{U}^{[k_1,k_2]}\ \phi_2 & \iff \exists 0 \leqslant h \leqslant k.\ \pi_\ell[h..] \models_\ell^{\mathcal{M}^N,\mathcal{L}} \phi_2 \\
 & & & \qquad \wedge \forall 0 \leqslant i < h.\ \pi_\ell[i..] \models \phi_1
\end{array}
$$

Figure 7.5: Local Formulas: Satisfaction relation

For the large part of operators, the definition of $\models$ (Fig. 7.4) is straightforward. Let $\pi$ be a *global computation*. We have that $\pi$ satisfies $\mathsf{true}$, $\pi$ satisfies $\Phi_1 \wedge \Phi_2$ if and only if both $\Phi_1$ and $\Phi_2$ are satisfied by $\pi$, while $\pi$ satisfies $\neg\Phi$ if and only if it does not satisfy $\Phi$. Temporal formula $\mathcal{X}\ \Phi$ is satisfied by $\pi$ if the computation starting from step 1 satisfies $\Phi$. Finally, $\pi$ satisfies $\Phi_1\ \mathcal{U}^{[k_1,k_2]}\ \Phi_2$ if and only if there exists an index $h \in [k_1, k_2]$ such that $\pi[h..]$ satisfies $\Phi_2$ and for any index $i$ less then $h$, $\pi[i..]$ satisfies $\Phi_1$.

An interesting case is represented by $\%[\phi] \bowtie p$. This formula is satisfied by a global computation $\pi$ if and only if the fraction of *agents* in $\pi$ having a *local computation* satisfying (local formula) $\phi$ is $\bowtie p$. Local computations of each single agent are obtained by considering $\pi \downarrow i$, the projection of $\pi$ on the index $i$, while $\#\pi[0]$ indicates the number of agents operating in the first state of the computation.

The definition of $\models_\ell$ is standard and is similar to what has been already discussed for the global case. The only case that deserves a bit of attention is the satisfaction of an atomic predicate $\mu$: a local path $\pi_\ell$ satisfies $\mu$ if and only if the state of the agent at the beginning of $\pi_\ell$ satisfies $\mu$, that is $\mu(\pi_\ell[0]) = \top$.

**Properties of Flock Scenario.** Once we defined GLoTL formalism, we can now use it for specifying some properties of the flock scenario. As we saw in Equation 7.1, we have already considered the formula $\Phi_{close}$, which requires that the fraction of agents *next to* the centre of mass should be greater than 75%. We can consider it a standard *global formula* that could be specified using an existing formalism. However, this formula could be too restrictive. The shown property is verified when *at the same time* 75% of the agents are next to the centre of mass. We can consider a different property where we require that 75% of the agents will be next to the centre of mass in the next $k$ time units:

$$\Phi_{ec} = \%[\Diamond^{[0,k]}\mu_{dist}] > 0.75 \tag{7.2}$$

The main global property that one is interested in monitoring is that *eventually* within $k_1$ steps, a configuration is reached where the formula of Equation 7.2 will be satisfied for at least $k_2$ steps. This property is rendered as follows:

$$\Phi_{eclose} = \Diamond^{[0,k_1]}\Box^{[0,k_2]}\Phi_{ec} \tag{7.3}$$

## 7.2 Monitoring GLoTL formulas

Following the same approach seen in [128] with the GLoTL model-checking, this section presents a monitoring algorithm that can be used to check if a YODA computation satisfies or not the proposed properties. The proposed algorithm is based on the classic idea that each formula is interpreted as *finite automaton*. For each computation step, the said automaton evolves until either an *accepting state* or a *rejecting state* is reached.

Given a local formula $\phi$ and an agent store $\sigma$, the function $\mathsf{after}_\mathsf{L}(\phi, \langle S, A\rangle)$, defined in Figure 7.6, yields the formula that a local computation starting with $\sigma$ must satisfy one step to fulfil $\phi$.

$$\mathsf{after}_\mathsf{L}(\mathsf{true}, \langle S, A\rangle) = \mathsf{true} \qquad \mathsf{after}_\mathsf{L}(\mathsf{false}, \langle S, A\rangle) = \mathsf{false}$$

$$\mathsf{after}_\mathsf{L}(\mu, \langle S, A\rangle) = \begin{cases} \mathsf{true} & \text{if } \mu(\langle S, A\rangle) = \top \\ \mathsf{false} & \text{if } \mu(\langle S, A\rangle) = \bot \end{cases}$$

$$\mathsf{after}_\mathsf{L}(\phi_1 \wedge \phi_2, \langle S, A\rangle) = \mathsf{after}_\mathsf{L}(\phi_1, \langle S, A\rangle) \wedge \mathsf{after}_\mathsf{L}(\phi_2, \langle S, A\rangle)$$

$$\mathsf{after}_\mathsf{L}(\phi_1 \vee \phi_2, \langle S, A\rangle) = \mathsf{after}_\mathsf{L}(\phi_1, \langle S, A\rangle) \vee \mathsf{after}_\mathsf{L}(\phi_2, \langle S, A\rangle)$$

$$\mathsf{after}_\mathsf{L}(\neg\phi, \langle S, A\rangle) = \neg\mathsf{after}_\mathsf{L}(\phi, \langle S, A\rangle) \qquad \mathsf{after}_\mathsf{L}(\mathcal{X}\ \phi, \langle S, A\rangle) = \phi$$

$$\begin{aligned}&\mathsf{after}_\mathsf{L}(\phi_1\ \mathcal{U}^{[k_1+1,k_2]}\ \phi_2, \langle S, A\rangle) = \\ &\qquad \mathsf{after}_\mathsf{L}(\phi_1, \langle S, A\rangle) \wedge \mathsf{after}_\mathsf{L}(\phi_1\ \mathcal{U}^{[k_1,k_2-1]}\ \phi_2, \langle S, A\rangle)\end{aligned}$$

$$\begin{aligned}&\mathsf{after}_\mathsf{L}(\phi_1\ \mathcal{U}^{[0,k_2+1]}\ \phi_2, \langle S, A\rangle) = \\ &\qquad (\mathsf{after}_\mathsf{L}(\phi_1, \langle S, A\rangle) \wedge \mathsf{after}_\mathsf{L}(\phi_1\ \mathcal{U}^{[0,k_2]}\ \phi_2, \langle S, A\rangle)) \\ &\qquad \vee\ \mathsf{after}_\mathsf{L}(\phi_2, \langle S, A\rangle)\end{aligned}$$

$$\mathsf{after}_\mathsf{L}(\phi_1\ \mathcal{U}^{[0,0]}\ \phi_2, \langle S, A\rangle) = \mathsf{after}_\mathsf{L}(\phi_2, \langle S, A\rangle)$$

Figure 7.6: Function $\mathsf{after}_\mathsf{L}$

**Theorem 1.** *For any local computation $\pi_\ell$ and for any local formula $\phi$:*

$$\pi_\ell \models_\ell \phi \qquad \Longleftrightarrow \qquad \pi_\ell[1..] \models_\ell \mathsf{after}_\mathsf{L}(\phi, \pi_\ell[0])$$

*Proof.* The proof proceeds by induction on the syntax of $\phi$.

**Base of Induction** We can distinguish two cases:

- ($\phi = \mathsf{true}$) The statement follows directly from the fact that for any $\pi_\ell$, $\pi_\ell \models \mathsf{true}$ and that for any $A$, $\mathsf{after_L}(\mathsf{true}, A) = \mathsf{true}$.

- ($\phi = \mu$) We can observe that if $\mu(A) = \mathsf{true}$, $\mathsf{after_L}(\mu, A) = \mathsf{true}$, while $\mathsf{after_L}(\mu, A) = \mathsf{false}$ whenever $\mu(A) = \bot$. Hence:

$$\begin{aligned}
\pi_\ell \models \phi &\Leftrightarrow \mu(\pi_\ell[0]) = \top \\
&\Leftrightarrow \mathsf{after_L}(\mu, A) = \mathsf{true} \\
&\Leftrightarrow \pi_\ell[1..] \models \mathsf{after_L}(\mu, A)
\end{aligned}$$

**Inductive Hypothesis** Let us assume that for $\phi_1$ and $\phi_2$ holds that

$$\pi_\ell \models_\ell \phi_i \qquad \Longleftrightarrow \qquad \pi_\ell[1..] \models_\ell \mathsf{after_L}(\phi_i, \pi_\ell[0])$$

**Inductive Step** The following cases must be considered:

- ($\neg\phi_1$) We have that:

$$\begin{aligned}
\pi_\ell \models \neg\phi_1 &\Leftrightarrow \pi_\ell \not\models \phi_1 \\
&\Leftrightarrow \pi_\ell[1..] \not\models \mathsf{after_L}(\phi_1, \pi_\ell[0]) \\
&\Leftrightarrow \pi_\ell[1..] \models \neg\mathsf{after_L}(\phi_1, \pi_\ell[0]) \\
&\Leftrightarrow \pi_\ell[1..] \models \mathsf{after_L}(\neg\phi_1, \pi_\ell[0])
\end{aligned}$$

- ($\phi_1 \wedge \phi_2$)

$$\begin{aligned}
\pi_\ell \models \phi_1 \wedge \phi_1 &\Leftrightarrow (\pi_\ell \models \phi_1) \\
&\quad\ \text{and } (\pi_\ell \models \phi_2) \\
&\Leftrightarrow \pi_\ell[1..] \models \mathsf{after_L}(\phi_1, \pi_\ell[0]) \\
&\quad\ \text{and } \pi_\ell[1..] \models \mathsf{after_L}(\phi_2, \pi_\ell[0]) \\
&\Leftrightarrow \pi_\ell[1..] \models \mathsf{after_L}(\phi_1, \pi_\ell[0]) \wedge \mathsf{after_L}(\phi_2, \pi_\ell[0]) \\
&\Leftrightarrow \pi_\ell[1..] \models \mathsf{after_L}(\phi_1 \wedge \phi_2, \pi_\ell[0])
\end{aligned}$$

- ($\mathcal{X}\ \phi_1$)

$$\begin{aligned}
\pi_\ell \models \mathcal{X}\ \phi_1 &\Leftrightarrow \pi_\ell[1..] \models \phi_1 \\
&\Leftrightarrow \pi_\ell[1..] \models \mathsf{after_L}(\mathcal{X}\ \phi_1, \pi_\ell[0])
\end{aligned}$$

- ($\phi_1\ \mathcal{U}^{[k_1,k_2]}\ \phi_2$) We have to distinguish three cases: $k_1 > 0$, $k_1 = 0$ and $k_2 > 0$, and $k_1 = k_2 = 0$.

  If $k_1 > 0$ we have that $\pi_\ell \models \phi_1\ \mathcal{U}^{[k_1,k_2]}\ \phi_2$ if and only if $\pi_\ell \models \phi_1$ and $\pi_\ell[1..] \models \phi_1\ \mathcal{U}^{[k_1-1,k_2-1]}\ \phi_2$. If $k_1 = 0$ and $k_2 > 0$ we have that $\pi_\ell \models \phi_1\ \mathcal{U}^{[k_1,k_2]}\ \phi_2$ if and only if either $\pi_\ell \models \phi_2$ or $\pi_\ell \models \phi_1$ and $\pi_\ell[1..] \models \phi_1\ \mathcal{U}^{[k_1-1,k_2-1]}\ \phi_2$. Finally, if $k_1 = k_2 = 0$ we have that $\pi_\ell \models \phi_1\ \mathcal{U}^{[k_1,k_2]}\ \phi_2$ if and only if $\pi_\ell \models \phi_2$. In all the cases the statement follows directly from inductive hypothesis and from the definition of $\mathsf{after}_\mathsf{L}$.

□

Similarly, this approach is used at *global level.* Given a global formula $\Phi$ and a system configuration $\alpha^*$, the function $\mathsf{after}_\mathsf{L}(\Phi, \alpha^*)$ yields the formula that a global computation starting with $\alpha^*$ must satisfy one step to fulfil $\Phi$, as we can see in Figure 7.7 An additional operator is introduced to handle the interplay between *local* and *global* formulas. The operator $\langle \phi_1, \ldots, \phi_N \rangle \bowtie p$ is used to assign each agent with a local formula and presents the following semantics:

$$\pi \models \langle \phi_1, \ldots, \phi_N \rangle \bowtie p \Leftrightarrow \frac{\left|\left\{ i | \pi \downarrow i \models_\ell^{\mathcal{M}^N,\mathcal{L}} \phi_i \right\}\right|}{N} \bowtie p$$

That said, we need to consider a second theorem that mirrors Theorem 1 but from the global point of view. This is illustrated below:

**Theorem 2.** *For any global computation $\pi$ and for any global formula $\Phi$:*

$$\pi \models \Phi \qquad \Longleftrightarrow \qquad \pi[1..] \models \mathsf{after}_\mathsf{G}(\Phi, \pi_\ell[0])$$

*Proof.* The proof proceeds by induction on the syntax of $\Phi$. All the cases are similar to the ones considered in the proof of Theorem 1. The only distinguishing cases are when $\Phi = \%[\phi] \bowtie p$ and $\langle \phi_1, \ldots, \phi_N \rangle \bowtie p$ that are direct consequences of Theorem 1. □

As presented in Algorithm 1, the monitoring algorithm is straightforward: the basic idea is that the configuration at each computational step is used to *transform* the monitored formula (by using function $\mathsf{after}_\mathsf{G}$) until either a formula that is *accepting* ($\Phi \uparrow$) or *rejecting*, ($\Phi \downarrow$) is reached.

Figure 7.8 defines said accepting and rejecting conditions, where $\Psi$ denotes both a local and a global formula. As expected, all rules follow the usual Boolean interpretation. Among them, the only interesting case is the one connected to the *auxiliary* formula $\langle \phi_1, \ldots, \phi_N \rangle \bowtie p$. This formula is in the *accepting* state whenever the fraction of local formulas $\phi_i$ that are

$$\mathsf{after_G}(\mathsf{true}, S) = \mathsf{true} \qquad \mathsf{after_G}(\mathsf{false}, , S) = \mathsf{false}$$

$$\mathsf{after_G}(\Phi_1 \wedge \Phi_2, S) = \mathsf{after_G}(\Phi_1, S) \wedge \mathsf{after_G}(\Phi_2, S)$$

$$\mathsf{after_G}(\Phi_1 \vee \Phi_2, S) = \mathsf{after_G}(\Phi_1, S) \vee \mathsf{after_G}(\Phi_2, S)$$

$$\mathsf{after_G}(\neg\Phi, S) = \neg\mathsf{after_G}(\Phi, S) \qquad \mathsf{after_G}(\mathcal{X}\ \Phi, S) = \Phi$$

$$\mathsf{after_G}(\Phi_1\ \mathcal{U}^{[k_1+1,k_2]}\ \Phi_2, , S) = \mathsf{after_G}(\Phi_1, S) \wedge \mathsf{after_G}(\Phi_1\ \mathcal{U}^{[k_1,k_2-1]}\ \Phi_2, S)$$

$$\mathsf{after_G}(\phi_1\ \mathcal{U}^{[0,k_2+1]}\ \phi_2, S) = \\ (\mathsf{after_G}(\Phi_1, S) \wedge \mathsf{after_G}(\Phi_1\ \mathcal{U}^{[0,k_2]}\ \Phi_2, S)) \vee\ \mathsf{after_G}(\Phi_2, S)$$

$$\mathsf{after_G}(\Phi_1\ \mathcal{U}^{[0,0]}\ \Phi_2, S) = \mathsf{after_G}(\phi_2, S)$$

$$\mathsf{after_G}(\%[\phi] \bowtie, S) = \langle \mathsf{after_L}(\phi, S \downarrow 1), \ldots, \mathsf{after_L}(\phi, S \downarrow |S|) \rangle \bowtie p$$

$$\mathsf{after_G}(\langle \phi_1, \ldots, \phi_N \rangle \bowtie p, S) = \langle \mathsf{after_L}(\phi_1, S \downarrow 1), \ldots, \mathsf{after_L}(\phi_N, S \downarrow N) \rangle \bowtie p$$

Figure 7.7: Operational Semantics of Global Formulas

**Algorithm 1** Monitoring GLoTL formulas

1: **function** MONITOR($i, \pi, \Phi$)
2:     **if** $\Phi \uparrow$ **then**
3:         **return** 1
4:     **end if**
5:     **if** $\Phi \downarrow$ **then**
6:         **return** 0
7:     **end if**
8:     **return** MONITOR($i + 1, \pi, \mathsf{after}(\Phi, \pi[i])$)
9: **end function**

$$\frac{}{\mathsf{true}\uparrow} \qquad \frac{}{\mathsf{false}\downarrow} \qquad \frac{\Psi_i \uparrow}{(\Psi_1 \vee \Psi_2)\uparrow} \qquad \frac{\Psi_1 \downarrow \quad \Psi_2 \downarrow}{(\Psi_1 \wedge \Psi_2)\downarrow}$$

$$\frac{\Psi_i \downarrow}{(\Psi_1 \wedge \Psi_2)\downarrow} \qquad \frac{\Psi_1 \uparrow \quad \Psi_2 \uparrow}{(\Psi_1 \vee \Psi_2)\uparrow} \qquad \frac{\Psi \downarrow}{(\neg\Psi)\uparrow} \qquad \frac{\Psi \uparrow}{(\neg\Psi)\downarrow}$$

$$\frac{\frac{|\{i|\phi_i\uparrow\}|}{N} \bowtie p}{\langle\phi_1,\ldots,\phi_N\rangle \bowtie p \uparrow} \qquad \frac{\frac{|\{i|\phi_i\downarrow\}|}{N} \overline{\bowtie}\, 1-p}{\langle\phi_1,\ldots,\phi_N\rangle \bowtie p \downarrow}$$

Figure 7.8: Acceptance and Rejection Criteria

*accepting* is $\bowtie p$. On the other hand, the same formula is *rejecting* when the fraction of local formulas $\phi_i$ that are rejecting is $\overline{\bowtie}(1-p)$, where $\overline{\bowtie}$ indicates the opposite relation of $\bowtie$.

The following theorem guarantees that for any formula $\Phi$ and global computation $\pi$ the monitoring procedure terminates with the correct value by performing at most $\mathsf{horizon}(\Phi)$ steps.

**Theorem 3.** *For any global path $\pi$ and global formula $\Phi$ the following properties are verified:*

- *MONITOR$(i,\pi,\Phi) = 1$ if and only if $\pi[i..] \models \Phi$;*
- *MONITOR$(i,\pi,\Phi)$ terminates in at most $\mathsf{horizon}(\Phi)$ steps.*

*Proof.* The proof proceeds easily by induction on the syntax of $\Phi$ by relying on on Theorem 1 and Theorem 2 and by observing that for any $\Phi$ (resp. $\phi$) and $\alpha^*$ (resp. $\alpha$) $\mathsf{horizon}(\Phi) > \mathsf{horizon}(\mathsf{after}_\mathsf{G}(\Phi,\alpha^*))$ (resp. $\mathsf{horizon}(\phi) > \mathsf{horizon}(\mathsf{after}_\mathsf{L}(\phi,\alpha)))$. □

The previously seen algorithm can reach both *offline* and *online* mode. This means that, to make a step forward, we need to consider only the *next* observed configuration. The algorithm has been integrated in Sibilla for supporting the analysis of YODA model. Estimating the probability that the considered properties are satisfied at a given step $t$ is possible thanks to the integration with the Sibilla model-checker.

Using the proposed approach, we can study the *global* and *local* aggregation of a flock. In the first case, a global aggregation is defined when a given fraction of agents are simultaneously next to the centre of mass and remain aggregated for a given number of steps. Using GLoTL notation, we can express this formula as follows:

$$\Phi_{ga} = \Diamond^{[0,k]}\Box^{[0,k_2]}(\%[\mu_{close}] > 0.9)$$

On the other hand, we need to focus on the behaviour of single agents when *local aggregation* is considered. As shown by the following formula, we specify that an agent eventually reaches the centre of mass and remains in its surroundings for a given number of steps.

$$\phi_{ls} = \Diamond^{[0,k_1]}\Box^{[0,k_2]}\mu_{close}{>}0.9)$$

The *local aggregation* property is reached when at least 90% of the agents satisfy formula $\phi_{ls}$:

$$\Phi_{la} = \Diamond^{[0,k]}(\%[\mu_{close}] > 0.9)$$

We report the results of the performed analyses in Figure 7.9. The analyses consider the number of agents ranging from $N = 5$ to $N = 50$.

## 7.3 Monitoring the Other Scenarios

This section shows how GLoTL can be used to monitor the scenarios introduced in Chapter 5.

### 7.3.1 Monitoring FinderBot

We recall that the goal of a FinderBot, presented in Section 5.2, is to follow a random walk until the goal area is reached and spread its position to other participants to let them move towards the target.

When an agent has reached the target, its attribute `found` is set to `true`. Hence, the predicate $\mu_{found}$ defined below is satisfied by the agents that are inside the goal area:

$$\mu_{found}(A) = A(found) == true$$

At the global level, we are interested in monitoring the fraction of agents that satisfy the predicate $\mu_{found}$. More specifically, we want to check if a given fraction of agents will eventually reach the goal area within a given time. This property can be formalised in GLoTL as follows:

$$\Phi_{gf} = \Diamond^{[0,k]}(\%[\mu_{found}] > 0.5)$$

In Figure 7.10, the result of the monitoring of $\Phi_{gf}$ is reported when we let the number of agents vary from 5 to 20 and the parameter $k$ range from 0 to 30. It is easy to see that the higher the number of agents, the higher the probability of reaching the goal area in $k$ steps.

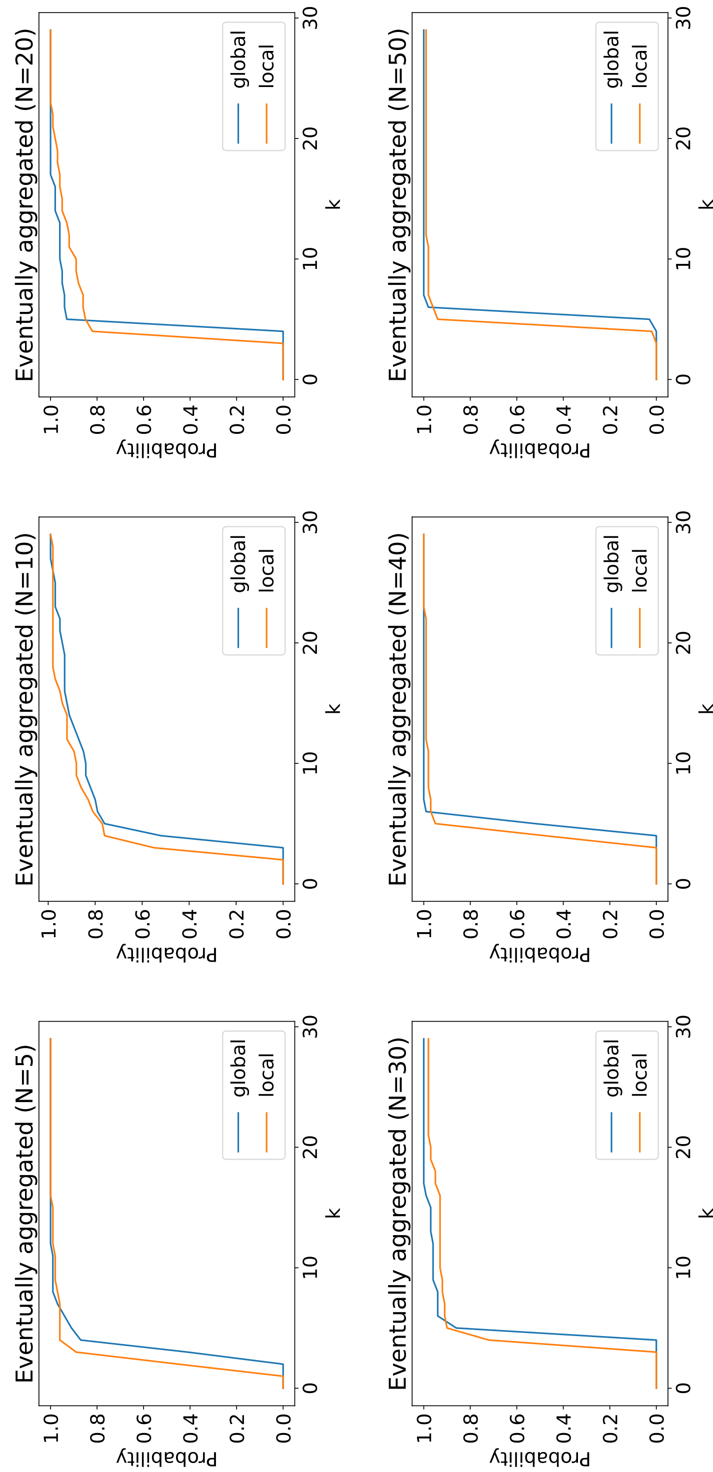



Figure 7.9: Global and Local Aggregation of Flock

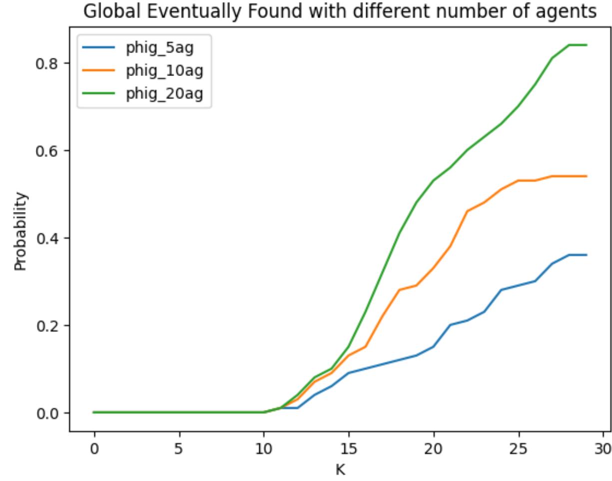



Figure 7.10: Global property applied to various cases of FinderBot

### 7.3.2 Monitoring SEIR

A SEIR model regards the evolution of an agent that can get infected and recover from the illness. This model's version has been introduced in Section 5.3.

For this scenario, we are interested in specifying properties regarding an agent's status, namely whether it is healthy (susceptible or recovered). The predicate $\mu_h$ defined below is satisfied only when the attribute `state` is either equal to $\mathsf{S}$ or $\mathsf{R}$:

$$\mu_s(A) = (A(state) == \mathsf{S})\&\&(A(state) == \mathsf{R})$$

We want to monitor the fraction of healthy agents at a given time step $k_1$ exceeds a threshold (0.5). This formula can be expressed as follows:

$$\Phi_{gh} = \mathcal{X}^k(\%[\mu_h] > 0.5)$$

where $\mathcal{X}^k\ \Phi$ is equivalent to $\Phi$ when $k = 0$, and to $\mathcal{X}\ \mathcal{X}^{k-1}\ \Phi$ when $k > 0$.

The formula $\Phi_{gh}$ is monitored in Figure 7.11. As seen in Section 5.3, the system considers 90% of susceptible individuals and 10% of infected ones. In the image, we consider 100, 500, and 1000 operating agents. The figure shows a pit around the 5th step. Then, at 10 time units, the graph stabilises itself and the property is always satisfied, at each time step. We can also see that, with more agents, there are less perturbations in the healthy population.

A more elaborated property could be used to amount the fraction of agents that, after $k$ steps, *remain healthy* for at least $k_2$ time units. This can be rendered in our logic in the following way:

$$\phi_{lh} = \Box^{[0,k_2]}\mu_h$$

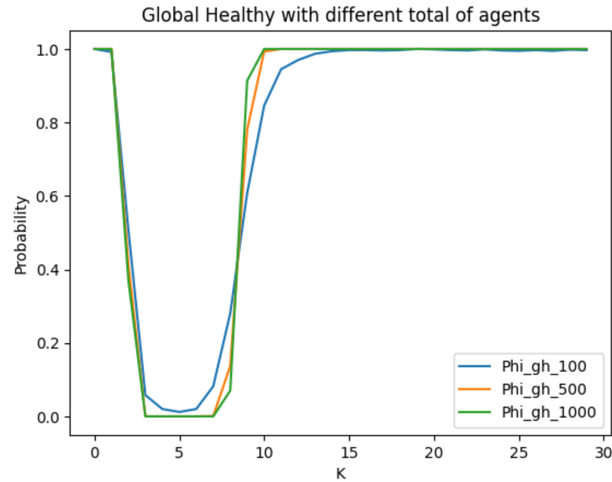



Figure 7.11: Results of Monitoring Formula $\Phi_{gh}$

$$\Phi_{glh} = \mathcal{X}^k(\%[\phi_{lh}] > 0.3)$$

The results of the monitoring of formula $\Phi_{glh}$ are shown in Figure 7.12. With this property, we want to know the probability of being healthy for a determined amount of time, differently from the previous one which considers only the atomic healthy predicate. The infection usually happens at the beginning of the simulation, thus the probability of having a healthy population for a given timespan is lower. In the image, we see that this probability increases after 7 steps, and becomes certain after 10 steps.

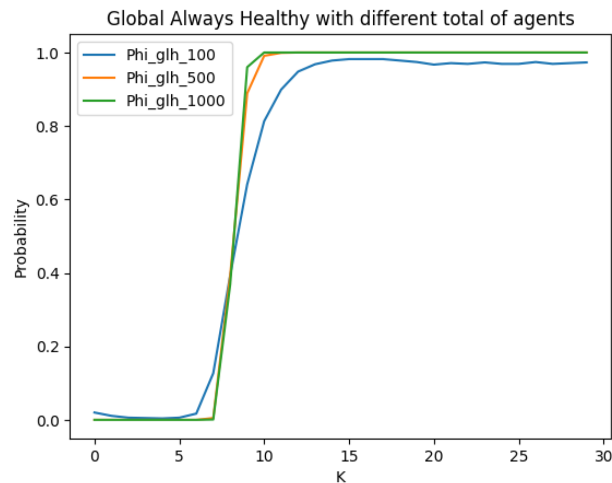



Figure 7.12: Results of Monitoring Formula $\Phi_{glh}$

The two properties above give us a measure of the fraction of agents that

are or remain *healthy*. However, one could also be interested in considering the fraction of agents that will be infected and will no recover return healthy for a while. The following formula is satisfied by the agents that will be infected within $k_1$ time steps and that remain infected for $k_2$ time steps:

$$\mu_i(\alpha) = A(state) == \mathsf{I}$$

$$\phi_{ldi} = \Diamond^{[0,k_1]}(\Box^{[0,k_2]}\mu_i)$$

At a global level, we can monitor that after $k$ time steps, the fraction of agents satisfying $\phi_{ldi}$ is less than 25%:

$$\Phi_{gdi} = \mathcal{X}^k(\%[\phi_{ldi}] < 0.25)$$

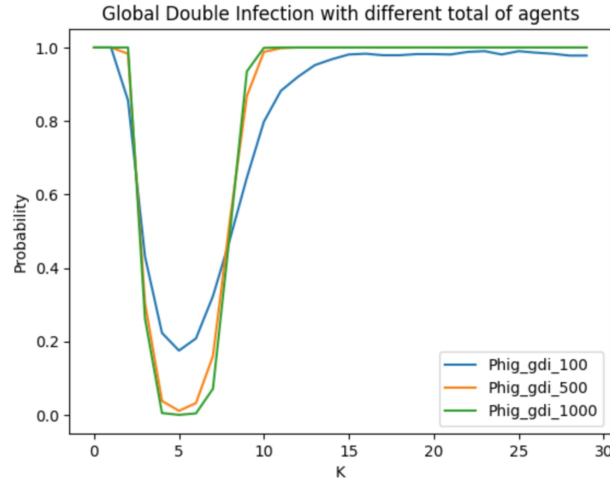



Figure 7.13: Results of Monitoring Formula $\Phi_{gdi}$

In Figure 7.13, we show the results of the monitoring performed on $\Phi_{gdi}$. The plot shows how the double infection is more severe at the 5th step. The pit means that more double infections occur around that time and the formula can not be satisfied. On the other hand, the property is verified for each scenario after the 10th time step.

### 7.3.3 Monitoring Red-Blue

In the Red-Blue scenario shown in Section 5.4, we expect that, eventually, the two parties reach a quorum of 50%, meaning that half of the agent will be blue, while the other half will be red. Considering this, we declare two atomic predicates, one related to the `redParty` attribute, and one for the

`blueParty`. These are verified if the corresponding model attribute is set to `true`. We declare them as follows:

$$\mu_{red}(A) = A(redParty) == true$$

$$\mu_{blue}(A) = A(blueParty) == true$$

Now we want to apply said atomic predicate to understand if eventually, after some time steps, a certain fraction of agents has turned red at a global level. This fraction should be included in a range, in this case between 40% and 60%. This property can be expressed in GLoTL by using the following formula:

$$\Phi_{bal} = \Diamond^{[0,k_1]}\%[\mu_{red}] \in [0.4, 0.6]$$

The formula is applied to the model specified in Listing 5.9. In Figure 7.14, we show the reached probabilities with different parameters. The first test is done using the base specification, while the second considers a total of 1000 agents (700 red and 300 blue ones). The third monitor shows if we have a changing rate of 0.1, and finally, the case where the distance tolerance is equal to 1.

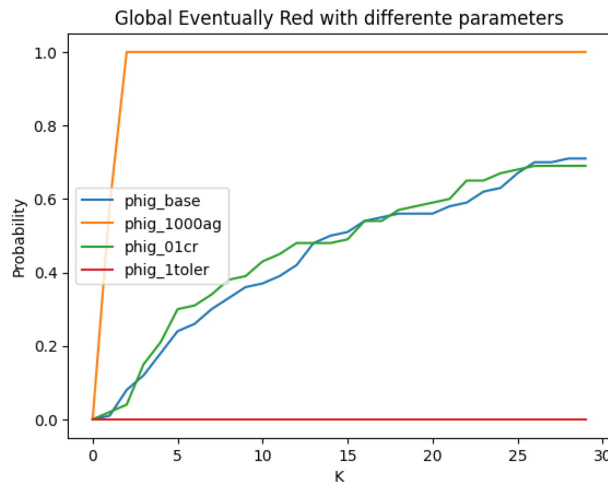



Figure 7.14: Global property applied to various cases of Red-Blue

Another property that one could be interested in verifying is the *global stability*. This means that the system is able to reach a configuration that is *balanced* and that it remains balanced for a while. This can be rendered as follows:

$$\Phi_{gs} = \Diamond^{[0,k_1]}\Box^{[0,k_2]}\%[\mu_{red}] \in [0.4, 0.6]$$

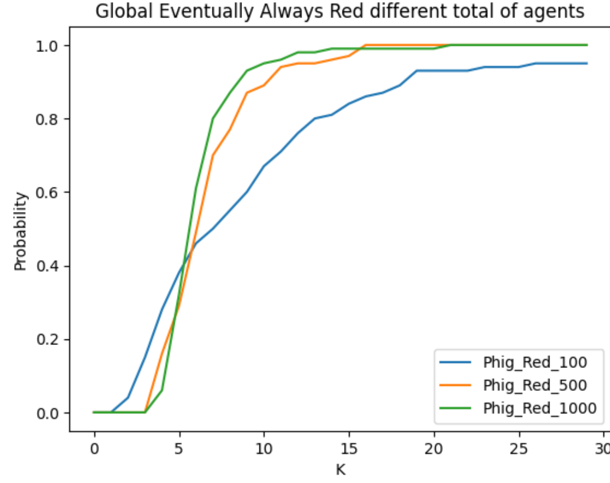



Figure 7.15: Results of Monitoring Formula $\Phi_{gs}$

In Figure 7.15, the results of monitoring the $\Phi_{gs}$ formula are presented. As we can see, the formula has been applied to three different scenarios: 100, 500, and 1000 total agents respectively. In each case, we respect the proportions of loaded agents, which are 70% red agents and 30% blue ones. The graph shows that, with a higher number of total agents, we can satisfy the formula in less time than the case with 100 agents.

Formula $\Phi_{gs}$describes a property of our system from a global point of view. However, another property that is of interest is the *local stability*. An agent is *local stable* if it remains of the same colour for a given number of steps. Local stability can be expressed as follows:

$$\phi_{ls} = \Diamond^{[0,k_1]}(\Box^{[0,k_2]}\mu_{blue} \vee \Box^{[0,k_2]}\mu_{red})$$

This formula states that, within $k_1$ steps, the agent will remain for $k_2$ steps of the same colour.

At the global level, we can check if that system can reach in at most $k_3$ steps a configuration where at least 75% are *locally stable*:

$$\Phi_{gls} = \Diamond^{[0,k_3]}\%[\phi_{ls}{>}0.75]$$

Figure 7.16 shows how the property $\Phi_{els}$ is verified. We assume that we have the same cases seen in the previous global formula. However, this time the property is easier to verify when we consider only 100 agents, while with more of them, we need more time. It is interesting to note that the property is hardly satisfied with the 1000 agents case where, within 30 time steps, the probability is less than 10%.

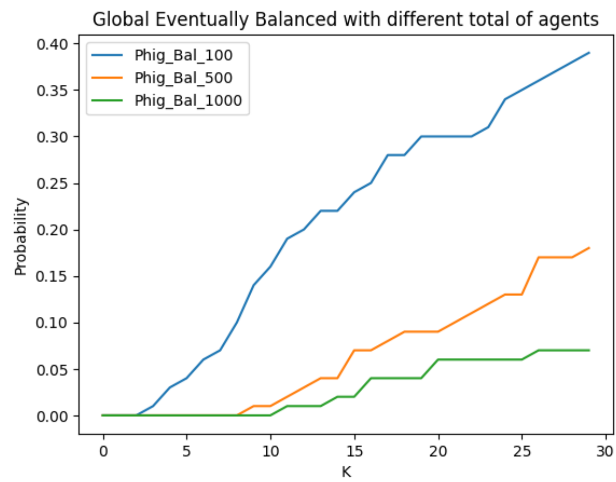



Figure 7.16: Results of Monitoring Formula $\Phi_{gls}$

# CHAPTER 8

# CONCLUSIONS

*Pass on*
*what you have learned.*

This thesis presented YODA, a framework for specifying and simulating Multi-Agent Systems. By taking inspiration from MAS languages that explicitly consider the environment, YODA combines environmental dynamics with agents' operations in order to have a better insight on unexpected results. Our proposal gives the environment a crucial role when designing MASs, by deputing it to send environmental data to the agents. This is useful when the agents may not have direct access to external information. In YODA, the environment is no longer a mere "world around", but an interacting entity.

In order to simulate YODA models, we relied on Sibilla, a Java-based simulation tool developed to reason and analyse collective systems. Sibilla has been designed to have a modular architecture, allowing the implementation of different languages about CAS. The architecture is divided into three main parts: the back-end, the front-end, and the runtime. The first one is responsible for performing the quantitative analyses feature in Sibilla, while the second component allows us to interact with the tool. The last part of the architecture permits the interactions between the back-end and the front-end. We also saw how we can work with Sibilla by exploiting two examples based on the Language of Population Model. YODA as been integrated in Sibilla by following the tool modular approach. We developed a suite of classes that are able to parse and execute a model that has been defined using YODA language constructs. We decided to include our language in Sibilla due to the characteristics of the tool. Indeed, during the integration of YODA, we did not have to worry about the implementation details of the simulator's

core. The core classes have been defined to simulate every available language, leaving to the language designer the duty of adding the language logic. We used a running scenario based on a group of robots moving inside an arena to demonstrate the integration of our language in Sibilla.

We applied YODA by specifying four different agent-based case scenarios: a flock behaviour, a group of rescuers, an epidemic scenario, and a consensus model. The first example was based on a group of birds (or boids, as called by Reynolds) moving in an environment without colliding with each other or distancing too much. The second example took inspiration from critical scenarios, like rescuing injured civilians, or agricultural ones, like weed infestations. It is represented by a team of finder agents looking for the target and notifying the others once the objective has been retrieved. The third example was inspired by the SEIR epidemic scenario. Here, a group of susceptible individuals got in contact with infected people. This scenario mimics the evolution of a transmittable illness. Finally, the last scenario showed how agents belonging to a party change sides when is around an higher number of members of its own party.

We developed a 3D visualiser called Sequit to represent these complex systems dynamically. The tool was used to simulate offline the agent traces and acts as a standalone, lightweight visualiser. We developed it using the Unity Game Engine: the choice relied on the advantages of using a well-structured solution. We explained Sequit architecture, divided into three main Game Object plus a set of utility classes, and the available features of the visualiser. We used the four scenarios of Chapter 5 to visualise the system's evolution dynamically, matching the right colour, depending on a chosen state. We also showed how we can visualise the simulation both using 3D and 2D environments.

Finally, we used the temporal logic framework GLoTL, which is available in Sibilla, to perform model-checking analyses. GLoTL is conceived to verify the local and global properties of complex systems. We applied the framework to YODA systems and analysed the proposed case studies. For each one of them, we proposed a set of local and global formulas to monitor YODA models and we verified them using the model-checking algorithm available in Sibilla. The satisfaction probabilities have been reported using Python plots.

## 8.1 Future Works

YODA framework and associated tools can be extended in several ways. As a first improvement, we would like to consider revising existing examples and adding new ones. For example, even if the Robotic Scenario of Chapter 4 has a probabilistic behaviour, allowing them to turn equally right or left, we want to expand the behavioural rules. The agents could perform more

complex actions or even sense the other agents to avoid colliding. Agents could implement strategic behaviours: as an example, if the robot finds itself surrounded by obstacles, it can try to go out of the *cul-de-sac* by going back. About new case studies, we can use YODA to define scenarios inspired by gaming ones [130]. For example, we can implement and simulate a scenario where two agents have to hit each other while moving in a linear arena. This example is inspired by the game *Artillery* [131]. Concerning gaming, it is also interesting to test the YODA framework in strategy videogames [132]. In these games, we have many agents acting independently after a particular order has been imparted. These entities should care also about the enclosing environment, which could change at any time. For this reason, we can imagine an application of YODA on a simulated battle scenario. Additionally, we claimed that YODA is conceived to be easily used by model designers. To validate this statement, we will ask a group of developers to specify a YODA model and give feedback on the usability of the language. Indeed, the authors developed the case studies presented in this thesis: external feedback could permit us to improve the language and make it more user-friendly.

In this thesis, we used a temporal logic framework to study the local and global properties of YODA models. However, these are a few of the possible properties we can assess through model-checking techniques. In realistic scenarios, perturbations may occur. These events affect the correct execution of the behavioural rules of a system, generating unexpected outcomes. Such issues are deeply analysed in [133] and [134], where perturbations are addressed at different levels. The authors of [133] measure a system's *adaptability* and *reliability*. An agent is considered adaptable if, after being perturbed, it can retrieve the initial condition in a reasonable amount of time. On the other hand, a program is reliable when it can maintain its main behaviour, even in the presence of said perturbations. In [134], the authors verify the *robustness* of CASs. This property deals with unknown events (uncertainties and perturbations) and refers to any measure of the capability of a program to tolerate said events and still complete its objective. In light of this, we think that these two frameworks can be applied to YODA systems. Currently, our systems are unconstrained, meaning that evolution steps are straightforward and data transmission doesn't consider error margin. To make it more realistic, we plan to revise the formalism to accept even perturbations and then apply the two frameworks to verify the related properties.

Another expansion of the framework could regard the introduction of *Reinforcement Learning* (RL) [135] techniques in a YODA model. RL is part of the machine learning techniques. It is based on functions allowing the agent to maximize an overall reward. Each action is associated with a value (the reward), and the agent tends to perform these actions more frequently to reach a higher cumulative reward. In case there are multiple agents, this technique

is referred to as *Multi-Agent Reinforcement Learning* (MARL) [136]. In the current version, YODA agents are tied to the design's set of behavioural rules. Their arbitrariness is related to a group of actions chosen on some probabilities, which are static and declared a priori. Because some scenarios may require strategic behaviour, we may need to define a behaviour that changes depending on the optimal decision. Thus, we can improve YODA behavioural syntax by defining the behaviour not as a set of probabilistic rules but as a group of rules chosen based on a reward function.

# PART I

# APPENDIX MATERIALS

# APPENDIX A

# LPM SPECIFICATIONS

In this appendix, we list the full specifications of the examples described in Section 3.3. We start with the *Repressilator* model, an auto-regulating genetic network, and then we move to the *Predator-Prey* scenario, where a pack of wolves has to hunt a flock of sheep. These specifications have been developed for the Special Issue of the Sibilla tool [16].

## A.1 Repressilator

```
param ln2 = 0.69314718056;
param beta = 0.2;
param alpha0 = 0.2164;
param alpha = 216.404;
param eff = 20;
param n = 2;
param KM = 40;
param tau_mRNA = 2;
param tau_prot = 10;
param ps_a = 0.5;
param ps_0 = 0.0005;
param t_ave = tau_mRNA / ln2;
param kd_mRNA = ln2 / tau_prot;
param kd_prot = ln2 / tau_mRNA;
param k_tl = eff / t_ave;
param a_tr = (ps_a -ps_0)*60;
param a0_tr = ps_0 * 60;

const startY = 20; /* Initial number of S agents */
const startVOID = 1; /* Initial number of I agents */

species VOID;  /* Fictitious species representing nothingness */

species PX; /*  protein produced by X */
species PY; /*  protein produced by Y */
species PZ; /*  protein produced by Z */
species X; /*  mRNA X (LacI) */
```

```
species Y; /*  mRNA Y (TetR) */
species Z; /*  mRNA Z (CI)*/

rule degradation_of_X_transcripts {
    X|VOID -[ kd_mRNA * #X ]-> VOID
}

rule degradation_of_Y_transcripts {
    Y|VOID -[ kd_mRNA * #Y ]-> VOID
}

rule degradation_of_Z_transcripts {
    Z|VOID -[ kd_mRNA * #Z ]-> VOID
}

rule translation_of_X {
    VOID -[ k_tl * #X ]-> VOID|PX
}

rule translation_of_Y {
    VOID -[ k_tl * #Y ]-> VOID|PY
}

rule translation_of_Z {
    VOID -[ k_tl * #Z ]-> VOID|PZ
}

rule degradation_of_X {
    PX|VOID -[ kd_prot * #PX ]-> VOID
}

rule degradation_of_Y {
    PY|VOID -[ kd_prot * #PY ]-> VOID
}

rule degradation_of_Z {
    PZ|VOID -[ kd_prot * #PZ ]-> VOID
}

rule transcription_of_X {
    VOID -[ a0_tr + ( ( a_tr * KM^(n) )/( KM^n + #PZ^n ) ) ]-> VOID|X
}

rule transcription_of_Y {
    VOID -[ a0_tr + ( ( a_tr * KM^(n) )/( KM^n + #PX^n ) ) ]-> VOID|Y
}

rule transcription_of_Z {
    VOID -[ a0_tr + ( ( a_tr * KM^(n) )/( KM^n + #PY^n ) ) ]-> VOID|Z
}

system initial = Y< startY >|VOID< startVOID >;
```

Listing A.1: Repressilator model in LPM

## A.2 Predator-Prey

```
const startW = 20;  /* Initial number of wolves */
const startS = 100; /* Initial number of sheeps */
const N = 7;        /* Width of the Grid */
```

```
const M = 7;        /* Length of the Grid */

param lambdaMovementS = 1.5; /* Rate of sheeps movement */
param lambdaMovementW = 1.2; /* Rate of wolves movement */
param lambdaMeet = 10.0;     /* Rate of wolves meeting with sheeps */
param eatingProb = 0.5 ;     /* Probability of wolf eating a sheep */

species W of [0,N]*[0,M];
species S of [0,N]*[0,M];

rule go_up_S for i in [0,N] and j in [0,M] when j<M-1{
    S[i,j] -[ lambdaMovementS * (1 - (%W[i,j+1])) ]-> S[i,j+1]
}

rule go_down_S for i in [0,N] and j in [0,M] when j>0{
    S[i,j] -[ lambdaMovementS * (1 - (%W[i,j-1])) ]-> S[i,j-1]
}

rule go_right_S for i in [0,N] and j in [0,M] when i<N-1{
    S[i,j] -[ lambdaMovementS * (1 - (%W[i+1,j])) ]-> S[i+1,j]
}

rule go_left_S for i in [0,N] and j in [0,M] when i>0{
    S[i,j] -[ lambdaMovementS * (1 - (%W[i-1,j])) ]-> S[i-1,j]
}

rule eating for i in [0,N] and j in [0,M]{
    W[i,j]|S[i,j] -[ (lambdaMeet*(%W[i,j])*(eatingProb)) ]-> W[i,j]
}

rule go_up_W for i in [0,N] and j in [0,M] when j<M-1{
    W[i,j] -[ 1+(lambdaMovementW * (%S[i,j+1])) ]-> W[i,j+1]
}

rule go_down_W for i in [0,N] and j in [0,M] when j>0{
    W[i,j] -[ 1+(lambdaMovementW * (%S[i,j-1])) ]-> W[i,j-1]
}

rule go_right_W for i in [0,N] and j in [0,M] when i<N-1{
    W[i,j] -[ 1+(lambdaMovementW * (%S[i+1,j])) ]-> W[i+1,j]
}

rule go_left_W for i in [0,N] and j in [0,M] when i>0{
    W[i,j] -[ 1+(lambdaMovementW * (%S[i-1,j])) ]-> W[i-1,j]
}

system start = W[4,4]<startW>|S[4,4]<startS>;
system start_surrounded = W[0,0]<10>|W[6,0]<10>|W[0,6]<10>|W[6,6]<10>|S
    [3,3]<100>;
```

Listing A.2: Predator-Prey model in LPM

# APPENDIX B

# YODA SPECIFICATION LANGUAGE

This appendix contains the totality of the YODA specification language that has been introduced in Section 4.2.

```
element                   ::= constantDeclaration
                            | parameterDeclaration
                            | typeDeclaration
                            | agentDeclaration
                            | sceneElementDeclaration
                            | systemDeclaration
                            | groupDeclaration
                            | measureDeclaration
                            | predicateDeclaration
                            | configurationDeclaration
                            | functionDeclaration

functionDeclaration       ::= 'let' name=ID '('(types+=type args+=ID
                                (',' types+=type args+=ID)*)? ')'
                                '=' expr
measureDeclaration        ::= 'measure' ID '=' expr
predicateDeclaration      ::= 'predicate' ID '=' expr
groupDeclaration          ::= 'group' ID '{' (ID (',' agents += ID)*)? '}'

constantDeclaration       ::= 'const' name=ID '=' value=expr ';'
parameterDeclaration      ::= 'param' name=ID '=' value=expr ';'
typeDeclaration           ::= 'type' typeName=ID '{'
                                (fields+=recordFieldDeclaration
                                (';' fields+=recordFieldDeclaration)*)? '}'
recordFieldDeclaration    ::= type name=ID
```

```
sceneElementDeclaration   ::=  'element' agentName=ID '='
                                 (elFeatAttr+=nameDeclaration ';')*
                               'end'

agentDeclaration          ::=  'agent' agentName=ID '='
                                 'state' ':'
                                   (agStateAttr+=nameDeclaration';')*
                                 'features' ':'
                                   (agFeatAttr+=nameDeclaration';')*
                                 'observations' ':'
                                   (agObsAttr+=nameDeclaration';')*
                                 'actions' ':' actionBody*
                                 'behaviour' ':' behaviourDeclaration
                               'end'
nameDeclaration           ::=  type name=ID '=' value=expr;
actionBody                ::=  actName=ID '[' (body+=attributeUpdate)* ']'
attributeUpdate           ::=  'let' names+=ID '=' values+=expr
                               ('and' names+=ID '=' values+=expr)*
                               'in' (body+=attributeUpdate)+
                               'endlet'
                            |  fieldName=ID '<-' value=expr ';'
                            |  'if' guard=expr 'then'
                               (thenBlock+=attributeUpdate)*
                               ('else' (elseBlock+=attributeUpdate)+)?
                               'endif'
behaviourDeclaration      ::=  ('when' cases+=ruleCase
                               ('orwhen' cases+=ruleCase)*
                               'otherwise')?
                               ('[' defaultcase+=weightedAction* ']')?
ruleCase                  ::=  guard=expr '->' '[' actions+=weightedAction* ']'
weightedAction            ::=  actionName=ID ':' weight=expr

systemDeclaration         ::=  'environment' ':'
                                 ('global' ':' (globAttr+=attributeUpdate)*)?
                                 ('sensing' ':'(agSensing+=agentAttributesUpdate)*)?
                                 ('dynamic' ':'(agDynamics+=agentAttributesUpdate)*)?
                               'end'
agentAttributesUpdate     ::=  agentName=ID '[' (updates += attributeUpdate)*']'

configurationDeclaration  ::=  'configuration' name=ID ':'
                                 (collectives += collectiveExpression)*
                               'end';
collectiveExpression      ::=  elementName=ID
                               '[' (init+=fieldAssignment ';')* ']'
                            |  'for' name=ID setOfValues 'do'
                               (body+=collectiveExpression)* 'endfor'
                            |  'if' guard=expr
                               'then' (thenCollective+=collectiveExpression)*
                               ('else' (elseCollective+=collectiveExpression)*)
                               'endif'
setOfValues               ::=  'from' from=expr 'to' to=expr
```

```
                 |  'in' '{' values+=expr (',' values+=expr) '}'
                 |  'sampled' (distinct='distinct')? size=expr
                    'time' 'from' generator=expr

expr            ::= INTEGER
                 |  REAL
                 |  'false'
                 |  'true'
                 |  reference=ID ('('(params+=expr
                    (',' params+=expr)*)? ')')?
                 |  '(' expr ')'
                 |  oper=('+'|'-') arg=expr
                 |  leftOp=expr oper=('+'|'-') rightOp=expr
                 |  leftOp=expr oper=('*'|'/') rightOp=expr
                 |  leftOp=expr oper=('%'|'//') rightOp=expr
                 |  leftOp=expr '^' rightOp=expr
                 |  '!' argument=expr
                 |  'sqrt' '(' argument=expr ')'
                 |  leftOp=expr oper=('&'|'&&') rightOp=expr
                 |  leftOp=expr oper=('|'|'||') rightOp=expr
                 |  leftOp=expr '->' rightOp=expr
                 |  leftOp=expr
                    oper=('<'|'<='|'=='|'!='|'>='|'>')
                    rightOp=expr
                 |  guardExpr=expr '?' thenBranch=expr
                    ':' elseBranch=expr
                 |  '[' fieldAssignment (',' fieldAssignment)* ']'
                 |  'U' '[' min=expr ',' max=expr ']'
                 |  'rnd'
                 |  'all' (groupName=ID)? ':' expr
                 |  'any' (groupName=ID)? ':' expr
                 |  'min' (groupName=ID)?
                    ('[' guard=expr ']' )? '.' value=expr
                 |  'max' (groupName=ID)?
                    ('[' guard=expr ']' )? '.' value=expr
                 |  'sum' (groupName=ID)?
                    ('[' guard=expr ']' )? '.' value=expr
                 |  'mean' (groupName=ID)?
                    ('[' guard=expr ']' )? '.' value=expr
                 |  '#' (groupName=ID)? ('[' guard=expr ']' )?
                 |  record=expr '.' fieldName =ID
                 |  'it.' ref=ID
                 |  'sin' '(' argument=expr ')'
                 |  'sinh' '(' argument=expr ')'
                 |  'asin' '(' argument=expr ')'
                 |  'cos' '(' argument=expr ')'
                 |  'cosh' '(' argument=expr ')'
                 |  'acos' '(' argument=expr ')'
                 |  'tan' '(' argument=expr ')'
                 |  'tanh' '(' argument=expr ')'
                 |  'atan' '(' argument=expr ')'
```

| | | |
|---|---|---|
| | \| | ‘`ceil`’ ‘`(`’ argument=expr ‘`)`’ |
| | \| | ‘`floor`’ ‘`(`’ argument=expr ‘`)`’ |
| | \| | ‘`abs`’ ‘`(`’ argument=expr ‘`)`’ |
| | \| | ‘`PI`’ |
| **fieldAssignment** | ::= | name=ID ‘=’ value=expr; |
| **type** | ::= | ‘`int`’ |
| | \| | ‘`real`’ |
| | \| | ‘`bool`’ |
| | \| | name=ID |

# BIBLIOGRAPHY

[1] Edward A. Lee. Cyber physical systems: Design challenges. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), 5-7 May 2008, Orlando, Florida, USA*, pages 363–369. IEEE Computer Society, 2008.

[2] Nilanjan Dey, Amira S. Ashour, Fuqian Shi, Simon James Fong, and João Manuel R. S. Tavares. Medical cyber-physical systems: A survey. *J. Medical Syst.*, 42(4):74:1–74:13, 2018.

[3] Anas AlMajali, Eric B. Rice, Arun Viswanathan, Kymie Tan, and Clifford Neuman. A systems approach to analysing cyber-physical threats in the smart grid. In *IEEE Fourth International Conference on Smart Grid Communications, SmartGridComm 2013, Vancouver, BC, Canada, October 21-24, 2013*, pages 456–461. IEEE, 2013.

[4] Matthias M. Hölzl, Axel Rauschmayer, and Martin Wirsing. Engineering of software-intensive systems: State of the art and research challenges. In Martin Wirsing, Jean-Pierre Banâtre, Matthias M. Hölzl, and Axel Rauschmayer, editors, *Software-Intensive Systems and New Computing Paradigms - Challenges and Visions*, volume 5380 of *Lecture Notes in Computer Science*, pages 1–44. Springer, 2008.

[5] David Sanderson, Nikolas Antzoulatos, Jack C. Chaplin, Dídac Busquets, Jeremy Pitt, Carl German, Alan Norbury, Emma Kelly, and Svetan M. Ratchev. Advanced manufacturing: An industrial application for collective adaptive systems. In *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASO Workshops 2015, Cambridge, MA, USA, September 21-25, 2015*, pages 61–67. IEEE Computer Society, 2015.

[6] Regina Frei and Giovanna Di Marzo Serugendo. Concepts in complexity engineering. *Int. J. Bio Inspired Comput.*, 3(2):123–139, 2011.

[7] Michael J. Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *Knowl. Eng. Rev.*, 10(2):115–152, 1995.

[8] Edmund H. Durfee, Victor R. Lesser, and Daniel D. Corkill. Trends in cooperative distributed problem solving. *IEEE Trans. Knowl. Data Eng.*, 1(1):63–83, 1989.

[9] Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *Auton. Agents Multi Agent Syst.*, 14(1):5–30, 2007.

[10] Rodney A Brooks. Achieving artificial intelligence through building robots. 1986.

[11] Karim Zeghal and Jacques Ferber. Craash- a coordinated collision avoidance system. *ONERA*, page 6, 1993.

[12] Rocco De Nicola, Luca Di Stefano, and Omar Inverso. Multi-agent systems with virtual stigmergy. *Sci. Comput. Program.*, 187:102345, 2020.

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

[14] Jens Claßen and James P. Delgrande. An account of intensional and extensional actions, and its application to belief, nondeterministic actions and fallible sensors. In Meghyn Bienvenu, Gerhard Lakemeyer, and Esra Erdem, editors, *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*, pages 194–204, 2021.

[15] Nicola Del Giudice, Lorenzo Matteucci, Michela Quadrini, Aniqa Rehman, and Michele Loreti. Sibilla: A tool for reasoning about collective systems. In Maurice H. ter Beek and Marjan Sirjani, editors, *Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, volume 13271 of *Lecture Notes in Computer Science*, pages 92–98. Springer, 2022.

[16] Nicola Del Giudice, Lorenzo Matteucci, Michela Quadrini, Aniqa Rehman, and Michele Loreti. Sibilla: A tool for reasoning about collective systems. *Science of Computer Programming*, page 103095, 2024.

[17] Joaquim RRA Martins and Andrew Ning. *Engineering design optimization*. Cambridge University Press, 2021.

[18] Gul Agha and Karl Palmskog. A survey of statistical model checking. *ACM Trans. Model. Comput. Simul.*, 28(1):6:1–6:39, 2018.

[19] Michael B Elowitz and Stanislas Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403(6767):335–338, 2000.

[20] Peter J Wangersky. Lotka-volterra population models. *Annual Review of Ecology and Systematics*, 9:189–218, 1978.

[21] Nicola Del Giudice and Michele Loreti. YODA: Yet anOther agent Description lAnguage. In Roberto Casadei, Elisabetta Di Nitto, Ilias Gerostathopoulos, Danilo Pianini, Ivana Dusparic, Timothy Wood, Phyllis R. Nelson, Evangelos Pournaras, Nelly Bencomo, Sebastian Götz, Christian Krupitzer, and Claudia Raibulet, editors, *IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion, ACSOS-C 2022, Virtual, CA, USA, September 19-23, 2022*, pages 82–87. IEEE, 2022.

[22] Nicola Del Giudice and Michele Loreti. Simulation and Analysis of YODA systems in Sibilla. In Review.

[23] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[24] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, 2009.

[25] Edmond Gjondrekaj, Michele Loreti, Rosario Pugliese, Francesco Tiezzi, Carlo Pinciroli, Manuele Brambilla, Mauro Birattari, and Marco Dorigo. Towards a formal verification methodology for collective robotic systems. In *International conference on formal engineering methods*, pages 54–70. Springer, 2012.

[26] 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.

[27] Robin R. Murphy, Jeffery Kravitz, Sam Stover, and Rahmat Shoureshi. Mobile robots in mine rescue and recovery. *IEEE Robotics Autom. Mag.*, 16(2):91–103, 2009.

[28] Flavio Corradini, Sara Pettinari, Barbara Re, Lorenzo Rossi, and Francesco Tiezzi. A BPMN-driven framework for Multi-Robot System development. *Robotics Auton. Syst.*, 160:104322, 2023.

[29] Michael Y. Li and James S. Muldowney. Global stability for the SEIR model in epidemiology. *Mathematical Biosciences*, 125(2):155–164, 1995.

[30] Abdollah Amirkhani and Amir Hossein Barshooi. Consensus in multi-agent systems: a review. *Artif. Intell. Rev.*, 55(5):3897–3935, 2022.

[31] Nicola Del Giudice, Federico Maria Cruciani, and Michele Loreti. Visualisation of collective systems with sequit and sibilla. In Ilaria Castellani and Francesco Tiezzi, editors, *Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings*, volume 14676 of *Lecture Notes in Computer Science*, pages 277–294. Springer, 2024.

[32] Nicola Del Giudice, Michele Loreti, Michela Quadrini, and Aniqa Rehman. Monitoring local and global properties of collective adaptive systems. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Rigorous Engineering of Collective Adaptive Systems - 12th International Symposium, ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceedings, Part II*, volume 15220 of *Lecture Notes in Computer Science*, pages 281–296. Springer, 2024.

[33] Roberta Calegari, Giovanni Ciatto, Viviana Mascardi, and Andrea Omicini. Logic-based technologies for multi-agent systems: a systematic literature review. *Auton. Agents Multi Agent Syst.*, 35(1):1, 2021.

[34] Nick Jennings and Michael Wooldridge. Software agents. *IEE review*, 42(1):17–20, 1996.

[35] Oren Etzioni, Neal Lesh, and Richard Segal. Building softbots for unix. In *Software Agents—Papers from the 1994 Spring Symposium (Technical Report SS–94–03)*, pages 9–16, 1994.

[36] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[37] Qinglin Guo and Ming Zhang. A novel approach for multi-agent-based intelligent manufacturing system. *Inf. Sci.*, 179(18):3079–3090, 2009.

[38] Stephen DJ McArthur, Euan M Davidson, Victoria M Catterson, Aris L Dimeas, Nikos D Hatziargyriou, Ferdinanda Ponci, and Toshihisa Funabashi. Multi-agent systems for power engineering applications—part i: Concepts, approaches, and technical challenges. *IEEE Transactions on Power systems*, 22(4):1743–1752, 2007.

[39] Fu-Shiung Hsieh and Jim-Bon Lin. Scheduling patients in hospitals based on multi-agent systems. In Moonis Ali, Jeng-Shyang Pan, Shyi-Ming Chen, and Mong-Fong Horng, editors, *Modern Advances in Applied Intelligence - 27th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2014, Kaohsiung, Taiwan, June 3-6, 2014, Proceedings, Part I*, volume 8481 of *Lecture Notes in Computer Science*, pages 32–42. Springer, 2014.

[40] Faten Khemakhem, Hamdi Ellouzi, Hela Ltifi, and Mounir Ben Ayed. Agent-based intelligent decision support systems: A systematic review. *IEEE Trans. Cogn. Dev. Syst.*, 14(1):20–34, 2022.

[41] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a bdi-architecture. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA, April 22-25, 1991*, pages 473–484. Morgan Kaufmann, 1991.

[42] Michael Bratman. Intention, plans, and practical reason. 1987.

[43] Anand S. Rao and Michael P. Georgeff. Decision procedures for BDI logics. *J. Log. Comput.*, 8(3):293–342, 1998.

[44] Alejandro Guerra-Hernández, Amal El Fallah Seghrouchni, and Henry Soldano. Learning in BDI multi-agent systems. In Jürgen Dix and João Alexandre Leite, editors, *Computational Logic in Multi-Agent Systems, 4th International Workshop, CLIMA IV, Fort Lauderdale, FL, USA, January 6-7, 2004, Revised Selected and Invited Papers*, volume 3259 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 2004.

[45] Patrick Maupin and A-L Jousselme. A general algebraic structure for situation analysis. In *2005 7th International Conference on Information Fusion*, volume 2, pages 9–pp. IEEE, 2005.

[46] Anne-Laure Jousselme and Patrick Maupin. Interpreted systems for situation analysis. In *10th International Conference on Information Fusion, FUSION 2007, Québec, Canada, July 9-12, 2007*, pages 1–11. IEEE, 2007.

[47] Danny Weyns and Tom Holvoet. A formal model for situated multi-agent systems. *Fundam. Informaticae*, 63(2-3):125–158, 2004.

[48] Danny Weyns, Giuseppe Vizzari, and Tom Holvoet. Environments for situated multi-agent systems: Beyond infrastructure. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for Multi-Agent Systems II, Second International Workshop, E4MAS 2005, Utrecht, The Netherlands, July 25, 2005, Selected Revised and Invited Papers*, volume 3830 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2005.

[49] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE–A FIPA-compliant agent framework. In *Proceedings of PAAM*, volume 99, page 33. London, 1999.

[50] Fabio Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE.* Wiley, 2007.

[51] Michael J. North, Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, Charles M. Macal, Mark J. Bragen, and Pam Sydelko. Complex adaptive systems modeling with Repast Simphony. *Complex Adapt. Syst. Model.*, 1:3, 2013.

[52] H. Van Dyke Parunak. Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation *by Michael North and Charles Macal. J. Artif. Soc. Soc. Simul.*, 10(4), 2007.

[53] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Catalin Balan. MASON: A multiagent simulation environment. *Simul.*, 81(7):517–527, 2005.

[54] Sean Luke, Robert Simon, Andrew T. Crooks, Haoliang Wang, Ermo Wei, David Freelan, Carmine Spagnuolo, Vittorio Scarano, Gennaro Cordasco, and Claudio Cioffi-Revilla. The MASON simulation toolkit: Past, present, and future. In Paul Davidsson and Harko Verhagen, editors, *Multi-Agent-Based Simulation XIX - 19th International Workshop, MABS 2018, Stockholm, Sweden, July 14, 2018, Revised Selected Papers*, volume 11463 of *Lecture Notes in Computer Science*, pages 75–86. Springer, 2018.

[55] Javier Palanca, Andrés Terrasa, Vicente Julián, and Carlos Carrascosa. SPADE 3: Supporting the New Generation of Multi-Agent Systems. *IEEE Access*, 8:182537–182549, 2020.

[56] Olivier Boissier, Rafael H. Bordini, Jomi Fred Hübner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with JaCaMo. *Sci. Comput. Program.*, 78(6):747–761, 2013.

[57] Olivier Boissier, Rafael H. Bordini, Jomi Fred Hübner, and Alessandro Ricci. Dimensions in programming multi-agent systems. *Knowl. Eng. Rev.*, 34:e2, 2019.

[58] Stefan Emrich, Sergej Suslov, and Florian Judex. Fully agent based modellings of epidemic spread using AnyLogic. In *Proc. eurosim*, pages 9–13, 2007.

[59] Jalal Joseph Possik, Simon Gorecki, Ali Asgary, Adriano O. Solis, Gregory Zacharewicz, Mohammadali Tofighi, Mohammad Ali Shafiee, Asad A. Merchant, Mehdi Aarabi, Abel Guimaraes, and Nazanin Nadri. A distributed simulation approach to integrate anylogic and unity for virtual reality applications: Case of COVID-19 modelling and training in a dialysis unit. In José M. Cecilia and Francisco J. Martinez, editors, *25th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2021, Valencia, Spain, September 27-29, 2021*, pages 1–7. IEEE, 2021.

[60] Andrew Begel and Eric Klopfer. Starlogo tng: An introduction to game development. *Journal of E-Learning*, 53(2007):146, 2007.

[61] Mitchel Resnick. Starlogo: An environment for decentralized modeling and decentralized thinking. In Michael J. Tauber, editor, *Conference on Human Factors in Computing Systems: Common Ground, CHI '96, Vancouver, BC, Canada, April 13-18, 1996, Conference Companion*, pages 11–12. ACM, 1996.

[62] Patrick Taillandier, Benoit Gaudou, Arnaud Grignard, Nghi Quang Huynh, Nicolas Marilleau, Philippe Caillou, Damien Philippon, and Alexis Drogoul. Building, composing and experimenting complex spatial models with the GAMA platform. *GeoInformatica*, 23(2):299–322, 2019.

[63] Philippe Caillou, Benoit Gaudou, Arnaud Grignard, Quang Chi Truong, and Patrick Taillandier. A simple-to-use BDI architecture for agent-based modeling and simulation. In Wander Jager, Rineke Verbrugge, Andreas Flache, Gert de Roo, Lex Hoogduin, and Charlotte K.

Hemelrijk, editors, *Advances in Social Simulation 2015 [Papers from the Conference of the European Social Simulation Association 2015, Groningen, The Netherlands, 14-18 September 2015]*, volume 528 of *Advances in Intelligent Systems and Computing*, pages 15–28. Springer, 2015.

[64] Nathan P. Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, Sendai, Japan, September 28 - October 2, 2004*, pages 2149–2154. IEEE, 2004.

[65] Carlos E. Agüero, Nate Koenig, Ian Chen, Hugo Boyer, Steven C. Peters, John M. Hsu, Brian P. Gerkey, Steffi Paepcke, Jose L. Rivero, Justin Manzo, Eric Krotkov, and Gill A. Pratt. Inside the virtual robotics challenge: Simulating real-time robotic disaster response. *IEEE Trans Autom. Sci. Eng.*, 12(2):494–506, 2015.

[66] Giorgio Audrito, Luigi Rapetta, and Gianluca Torta. Extensible 3d simulation of aggregated systems with FCPP. In Maurice H. ter Beek and Marjan Sirjani, editors, *Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, volume 13271 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 2022.

[67] Giorgio Audrito. FCPP: an efficient and extensible field calculus framework. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020, Washington, DC, USA, August 17-21, 2020*, pages 153–159. IEEE, 2020.

[68] Stefano Mariani and Andrea Omicini. Game engines to model MAS: A research roadmap. In Corrado Santoro, Fabrizio Messina, and Massimiliano De Benedetti, editors, *Proceedings of the 17th Workshop "From Objects to Agents" co-located with 18th European Agent Systems Summer School (EASSS 2016), Catania, Italy, July 29-30, 2016*, volume 1664 of *CEUR Workshop Proceedings*, pages 106–111. CEUR-WS.org, 2016.

[69] Sim Keng Wai, Cheah WaiShiang, Muhammad Asyraf bin Khairuddin, Yanti Rosmunie Binti Bujang, Rahmat Hidayat, and Celine Haren Paschal. Autonomous agents in 3d crowd simulation through bdi architecture. *JOIV: International Journal on Informatics Visualization*, 5(1):1–7, 2021.

[70] Michelangelo Diamanti and Hannes Högni Vilhjálmsson. Extending the menge crowd simulation framework: visual authoring in unity. In Carlos Martinho, João Dias, Joana Campos, and Dirk Heylen, editors, *IVA '22: ACM International Conference on Intelligent Virtual Agents, Faro, Portugal, September 6 - 9, 2022*, pages 30:1–30:3. ACM, 2022.

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

[72] Gianluca Aguzzi, Roberto Casadei, and Mirko Viroli. Macroswarm: A field-based compositional framework for swarm programming. In Sung-Shik Jongmans and Antónia Lopes, editors, *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings*, volume 13908 of *Lecture Notes in Computer Science*, pages 31–51. Springer, 2023.

[73] Rinde R. S. van Lon and Tom Holvoet. Rinsim: A simulator for collective adaptive systems in transportation and logistics. In *Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2012, Lyon, France, September 10-14, 2012*, pages 231–232. IEEE Computer Society, 2012.

[74] Luca Bortolussi, Rocco De Nicola, Vashti Galpin, Stephen Gilmore, Jane Hillston, Diego Latella, Michele Loreti, and Mieke Massink. CARMA: Collective Adaptive Resource-sharing Markovian Agents. In Nathalie Bertrand and Mirco Tribastone, editors, *Proceedings Thirteenth Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2015, London, UK, 11th-12th April 2015*, volume 194 of *EPTCS*, pages 16–31, 2015.

[75] Jane Hillston and Michele Loreti. CARMA eclipse plug-in: A tool supporting design and analysis of collective adaptive systems. In Gul Agha and Benny Van Houdt, editors, *Quantitative Evaluation of Systems - 13th International Conference, QEST 2016, Quebec City, QC, Canada, August 23-25, 2016, Proceedings*, volume 9826 of *Lecture Notes in Computer Science*, pages 167–171. Springer, 2016.

[76] Vashti Galpin, Natalia Zon, Pia Wilsdorf, and Stephen Gilmore. Mesoscopic modelling of pedestrian movement using carma and its tools. *ACM Trans. Model. Comput. Simul.*, 28(2):11:1–11:26, 2018.

[77] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: probabilistic symbolic model checker. In Tony Field, Peter G. Harrison, Jeremy T. Bradley, and Uli Harder, editors, *Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference, TOOLS 2002, London, UK, April 14-17, 2002, Proceedings*, volume 2324 of *Lecture Notes in Computer Science*, pages 200–204. Springer, 2002.

[78] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker storm. *Int. J. Softw. Tools Technol. Transf.*, 24(4):589–610, 2022.

[79] C. A. R. Hoare. Communicating sequential processes (reprint). *Commun. ACM*, 26(1):100–106, 1983.

[80] Jane Hillston. *A compositional approach to performance modelling*. PhD thesis, University of Edinburgh, UK, 1994.

[81] Marco Bernardo, Lorenzo Donatiello, and Roberto Gorrieri. A formal approach to the integration of performance aspects in the modeling and analysis of concurrent systems. *Inf. Comput.*, 144(2):83–154, 1998.

[82] Rocco De Nicola, Gian-Luigi Ferrari, Rosario Pugliese, and Francesco Tiezzi. A formal approach to the engineering of domain-specific distributed systems. *J. Log. Algebraic Methods Program.*, 111:100511, 2020.

[83] Matthew Hennessy. Process calculi for describing distributed systems. *J. Comput. Sci. Technol.*, 13(6):490, 1998.

[84] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gian-Luigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto, and Betti Venneri. The klaim project: Theory and practice. In Corrado Priami, editor, *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems, IST/FET International Workshop, GC 2003, Rovereto, Italy, February 9-14, 2003, Revised Papers*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer, 2003.

[85] Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Thudichum Vasconcelos, and Gianluigi Zavattaro. SCC: A service centered calculus. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna,*

*Austria, September 8-9, 2006, Proceedings*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.

[86] Rosario Pugliese and Francesco Tiezzi. A calculus for orchestration of web services. *J. Appl. Log.*, 10(1):2–31, 2012.

[87] Federica Ciocchetta and Jane Hillston. Bio-PEPA: An Extension of the Process Algebra PEPA for Biochemical Networks. In Nicola Cannata and Emanuela Merelli, editors, *Proceedings of the First Workshop "From Biology To Concurrency and back", FBTC@CONCUR 2007, Lisbon, Portugal, September 8, 2007*, volume 194 of *Electronic Notes in Theoretical Computer Science*, pages 103–117. Elsevier, 2007.

[88] Ludovica Luisa Vissat, Jane Hillston, Glenn Marion, and Matthew J Smith. MELA: modelling in ecology with location attributes. *arXiv preprint arXiv:1610.08171*, 2016.

[89] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *ACM Trans. Auton. Adapt. Syst.*, 9(2):7:1–7:29, 2014.

[90] Yehia Abd Alrahman, Rocco De Nicola, Michele Loreti, Francesco Tiezzi, and Roberto Vigo. A calculus for attribute-based communication. In Roger L. Wainwright, Juan Manuel Corchado, Alessio Bechini, and Jiman Hong, editors, *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1840–1845. ACM, 2015.

[91] Sheldon M Ross, John J Kelly, Roger J Sullivan, William James Perry, Donald Mercer, Ruth M Davis, Thomas Dell Washburn, Earl V Sager, Joseph B Boyce, and Vincent L Bristow. *Stochastic processes*, volume 2. Wiley New York, 1996.

[92] James R. Norris. *Markov chains*. Cambridge series in statistical and probabilistic mathematics. Cambridge University Press, 1998.

[93] Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer, 1996.

[94] Donald A Darling and AJF58908 Siegert. The first passage problem for a continuous markov process. *The Annals of Mathematical Statistics*, pages 624–639, 1953.

[95] Gregor von Bochmann. Finite state description of communication protocols. *Comput. Networks*, 2:361–372, 1978.

[96] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working with objects - the OOram software engineering method.* Manning, 1996.

[97] Kivy Team and other contributors. pyjnius. `https://github.com/kivy/pyjnius`.

[98] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. Jupyter notebooks - a publishing format for reproducible computational workflows. In Fernando Loizides and Birgit Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7-9, 2016*, pages 87–90. IOS Press, 2016.

[99] Ekaba Bisong. *Google Colaboratory*, pages 59–64. Apress, Berkeley, CA, 2019.

[100] Luca Bortolussi, Jane Hillston, Diego Latella, and Mieke Massink. Continuous approximation of collective system behaviour: A tutorial. *Perform. Evaluation*, 70(5):317–349, 2013.

[101] William J Anderson. *Continuous-time Markov chains: An applications-oriented approach.* Springer Science & Business Media, 2012.

[102] David Rogers. Random search and insect population models. *The Journal of Animal Ecology*, pages 369–383, 1972.

[103] Chengjun Sun and Ying-Hen Hsieh. Global analysis of an seir model with varying population size and vaccination. *Applied Mathematical Modelling*, 34(10):2685–2697, 2010.

[104] Joseph Y. Halpern and Yoram Moses. Knowledge and Common Knowledge in a Distributed Environment. *J. ACM*, 37(3):549–587, 1990.

[105] Anne-Laure Jousselme and Patrick Maupin. Interpreted systems for situation analysis. In *10th International Conference on Information Fusion, FUSION 2007, Québec, Canada, July 9-12, 2007*, pages 1–11. IEEE, 2007.

[106] Wiebe van der Hoek and Michael J. Wooldridge. Tractable multiagent planning for epistemic goals. In *The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings*, pages 1167–1174. ACM, 2002.

[107] Bruce A Craig and Peter P Sendi. Estimation of the transition matrix of a discrete-time markov chain. *Health economics*, 11(1):33–42, 2002.

[108] Michele Loreti and Jane Hillston. Modelling and analysis of collective adaptive systems with CARMA and its tools. In Marco Bernardo, Rocco De Nicola, and Jane Hillston, editors, *Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems - 16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2016, Bertinoro, Italy, June 20-24, 2016, Advanced Lectures*, volume 9700 of *Lecture Notes in Computer Science*, pages 83–119. Springer, 2016.

[109] Valentina Castiglioni, Michele Loreti, and Simone Tini. Stark: A tool for the analysis of CPSs robustness. *Sci. Comput. Program.*, 236:103134, 2024.

[110] Valentina Castiglioni, Michele Loreti, and Simone Tini. Measuring adaptability and reliability of large scale systems. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II*, volume 12477 of *Lecture Notes in Computer Science*, pages 380–396. Springer, 2020.

[111] Rocco De Nicola, Luca Di Stefano, and Omar Inverso. Multi-agent systems with virtual stigmergy. *Sci. Comput. Program.*, 187:102345, 2020.

[112] Sidney Redner. *A guide to first-passage processes.* Cambridge university press, 2001.

[113] Gregor V Bochmann. Finite state description of communication protocols. *Computer Networks (1976)*, 2(4-5):361–372, 1978.

[114] Rocco De Nicola, Luca Di Stefano, Omar Inverso, and Serenella Valiani. Modelling flocks of birds from the bottom up. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods,*

*Verification and Validation. Adaptation and Learning - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part III*, volume 13703 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2022.

[115] Yaroslav Vyklyuk, Mykhailo Manylich, Miroslav Škoda, Milan M Radovanović, and Marko D Petrović. Modeling and analysis of different scenarios for the spread of covid-19 by using the modified multi-agent systems–evidence from the selected countries. *Results in Physics*, 20:103662, 2021.

[116] Etienne Perron, Dinkar Vasudevan, and Milan Vojnovic. Using three states for binary consensus on complete graphs. In *INFOCOM 2009. 28th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro, Brazil*, pages 2527–2535. IEEE, 2009.

[117] Daniele Vernon-Bido, Andrew J. Collins, and John A. Sokolowski. Effective visualization in modeling & simulation. In Saikou Y. Diallo and Andreas Tolk, editors, *Proceedings of the 48th Annual Simulation Symposium, ANSS 2015, part of the 2015 Spring Simulation Multiconference, SpringSim '15, Alexandria, VA, USA, April 12-15, 2015*, pages 33–40. SCS/ACM, 2015.

[118] Jerry Banks and Leonardo Chwif. Warnings about simulation. *J. Simulation*, 5(4):279–291, 2011.

[119] Michele Loreti and Jane Hillston. Modelling and analysis of collective adaptive systems with CARMA and its tools. In Marco Bernardo, Rocco De Nicola, and Jane Hillston, editors, *Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems - 16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2016, Bertinoro, Italy, June 20-24, 2016, Advanced Lectures*, volume 9700 of *Lecture Notes in Computer Science*, pages 83–119. Springer, 2016.

[120] Mirella Santos Pessoa de Melo, José Gomes da Silva Neto, Pedro Jorge Lima da Silva, João Marcelo Xavier Natario Teixeira, and Veronica Teichrieb. Analysis and comparison of robotics 3d simulators. In *21st Symposium on Virtual and Augmented Reality, SVR 2019, Rio de Janeiro, Brazil, October 28-31, 2019*, pages 242–251. IEEE, 2019.

[121] Christopher Pettit, Ian Bishop, Victor Sposito, Jean-Philippe Aurambout, and Falak Sheth. Developing a multi-scale visualisation frame-

work for use in climate change response. *Landscape ecology*, 27:487–508, 2012.

[122] Sandro Tosi. *Matplotlib for Python developers*. Packt Publishing Ltd, 2009.

[123] Tony Parisi. *WebGL: up and running*. " O'Reilly Media, Inc.", 2012.

[124] Tom Shannon. *Unreal Engine 4 for design visualization: Developing stunning interactive visualizations, animations, and renderings*. Addison-Wesley Professional, 2017.

[125] Diego Latella, Michele Loreti, and Mieke Massink. On-the-fly PCTL fast mean-field approximated model-checking for self-organising coordination. *Sci. Comput. Program.*, 110:23–50, 2015.

[126] Luca Bortolussi, Roberta Lanciani, and Laura Nenzi. Model checking markov population models by stochastic approximations. *Inf. Comput.*, 262:189–220, 2018.

[127] Yehia Abd Alrahman, Giuseppe Perelli, and Nir Piterman. Reconfigurable interaction for MAS modelling. In Amal El Fallah Seghrouchni, Gita Sukthankar, Bo An, and Neil Yorke-Smith, editors, *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020*, pages 7–15. International Foundation for Autonomous Agents and Multiagent Systems, 2020.

[128] Michele Loreti and Aniqa Rehman. A logical framework for reasoning about local and global properties of collective systems. In Erika Ábrahám and Marco Paolieri, editors, *Quantitative Evaluation of Systems - 19th International Conference, QEST 2022, Warsaw, Poland, September 12-16, 2022, Proceedings*, volume 13479 of *Lecture Notes in Computer Science*, pages 133–149. Springer, 2022.

[129] Alexandre Donzé. On signal temporal logic. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *Lecture Notes in Computer Science*, pages 382–383. Springer, 2013.

[130] Carlos Marín-Lora, Miguel Chover, José Martínez Sotoca, and Luis A. García. A game engine to make games as multi-agent systems. *Adv. Eng. Softw.*, 140, 2020.

[131] Mike Forman. War 3. *Creative Computing*, 2(1):68, Jan-Feb 1976.

[132] Damon Daylamani Zad, Letitia B. Graham, and Ioannis Theoklitos Paraskevopoulos. Swarm intelligence for autonomous cooperative agents in battles for real-time strategy games. In *9th International Conference on Virtual Worlds and Games for Serious Applications, VS-Games 2017, Athens, Greece, September 6-8, 2017*, pages 39–46. IEEE Computer Society, 2017.

[133] Valentina Castiglioni, Michele Loreti, and Simone Tini. How adaptive and reliable is your program? In Kirstin Peters and Tim A. C. Willemse, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12719 of *Lecture Notes in Computer Science*, pages 60–79. Springer, 2021.

[134] Valentina Castiglioni, Michele Loreti, and Simone Tini. Stark: A software tool for the analysis of robustness in the unknown environment. In Sung-Shik Jongmans and Antónia Lopes, editors, *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings*, volume 13908 of *Lecture Notes in Computer Science*, pages 115–132. Springer, 2023.

[135] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *J. Artif. Intell. Res.*, 4:237–285, 1996.

[136] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Trans. Syst. Man Cybern. Part C*, 38(2):156–172, 2008.

# ACKNOWLEDGMENTS

Finally, it is time to conclude this journey. With these few lines, nearly four years of hard work come to an end. To be honest, I never thought that doing a PhD could have been a possible choice, however, here I am. A doctoral course changes you deeply, in the soul, it makes you different and more mature. It challenges you, and not everyone can face it. But it also permits discovering new things, making new friends, and understanding what you can really do. These few words go against the stereotypical idea of the doctorate, which usually is considered stressful, full of sacrifices, without joy, and continuous disorders. On the contrary, I never hid that my PhD has been exciting and, somehow, fun. Of course, though moments occurred, still, if I am here today, I have to say thanks to a group of people.

- **To Simona e Paolo**, my parents, for the patience and encouraging words they spoke in these challenging years;
- **To Michele**, my supervisor, for giving me this possibility and being a mentor in a brand new world;
- **To Barbara**, for welcoming a lot of strange ideas, with a lot of enthusiasm;
- **To Daniele and Tommaso**, my two best friends, for sharing a lot of good and bad moments, and drinking a lot of Hattori Hanzos;
- **To the Flying Jug**, for the incredible adventures, in both real and fantastic life;
- **To Lorenzo**, my academic brother, for the mutual support and all of the shenanigans;
- **To the Anger Office**, for the anger outbursts, chitchats, and laughs;

- **To the Nightsong**, for believing in me, helping me dissipate the mist, and showing me that true happiness should be reached starting from themselves.

To all of you and many more, I have to say “Thank you”, and always remember…

***May The Force be with You***