Universiteit Antwerpen

Towards an Agent-Based Modeling Platform with Precise Semantics

Tim Leys

Principal Adviser: Prof. Hans Vangheluwe

Assistant Supervisors: Dr. Simon Van Mierlo
Dr. Romain Franceschini

Dissertation Submitted in Januari 2020 to the
Department of Mathematics and Computer Science
of the Faculty of Sciences, University of Antwerp,
In partial fulfillment of the requirements
for the degree of Master of Science.

Ansymo
Antwerp Systems and Software Modelling

Contents

List of Figures

List of Tables

Abstract

Agent-based modelling and simulation (ABMS) is a powerful technique for modelling complex systems that consist of multiple interacting entities. In ABMS, each entity is represented as an autonomous agent that observes and acts on an environment. The field of ABMS, however, is very divided and, although most research groups have similar interpretations of the concepts in ABMS, a universal agreement on a definition of an agent-based model remains absent. This problem manifests itself in the available agent-based modelling and simulation tools, since no tool is available which has formal or even precise semantics. The lack of uniformity results in a field where models are created on an individual basis and reuse-ability is scarce. In this study, we tackle this problem by investigating how agent-based modelling is currently realized and propose a formalism with precise semantics that captures the essential characteristics of agent-based modelling. Additionally, we present a tool which implements the proposed formalism. This new tool, SARLforSIM, is evaluated against a selection of current agent-based modelling and simulation tools. We performed a comparative study by implementing the same population dynamics model in each of the tools and investigated how they realize the features of an agent-based model. Finally, we also investigated the reproducibility of the model in the different tools.

Acknowledgements

First of all, I would like to thank my promoter, Hans Vangheluwe. He has been my mentor for the last two years for both my master thesis and internship with the MSDL research group. His unwavering enthusiasm for research has been a great inspiration for me and I am very grateful that he gave me the opportunity to do my thesis with MSDL.

I also want to thank my supervisors, Simon Van Mierlo and Romain Franceshini, who where always available to answer my questions and to give advise. Their guidance has helped me grow a lot as a researcher and a person.

Finally, I want to thank my friends and family for their support throughout my study career.

Nederlandstalige Samenvatting

Agent-Based Modelling and Simulation (ABMS) is een krachtige techniek om complexe systemen te voorstellen als een collectie van autonome, interagerende, beslissing makende entiteiten [14, 52]. Deze entititeiten, of agents, zijn in staat om hun omgeving te observeren en acties uit te voeren Ook kunnen ze in interactie gaan met andere agents door het uitwisselen van berichten. Hoe agents reageren in bepaalde situaties wordt beschreven in het gedrag van de agent. Het gedrag van de agent wordt beinvloed door de gemaakte observaties en de informatie dat de agents onderling uitwisselen. Fabien Michel [58] noemt agent-based models artificiele micro werelden, waarvan het mogelijk is alle karacteristieken te controleren en om een serie experimenten te reproduceren zoals in een labratorium.

Het onderzoeksveld van agent-based modelling en simulatie is echter verdeeld. Verschillende onderzoeksgroepen uit verschillende domeinen werken met ABMS op een geisoleerde manier. Dit resulteerd in een onderzoeksveld waarin er geen eenduidigheid is omtrent een definitie van ABMS. Hoewel de meeste onderzoeksgroepen meestal overeenstemmen over de verschillende concepten binnen ABMS, is het niet triviaal om modelen te classificeren als agent-based of niet. Dit wordt onderstreept door de verschillende definities, gegeven door Charles M. Macal [51], die nodig zijn om het brede spectrum van agent based modellen te beschrijven. Het gebrek aan eenduidigheid manifesteert zich ook in de beschikbare tools waarin men agent-based modellen kan beschrijven en simuleren. Tools zoals NetLogo [79] en MASON [49] hebben een verschillende interpretatie over de precieze semantiek van een agent. Bovendien houden deze tools zich niet aan een precieze semantiek, wat het analyzeren van model aanzienlijk complexer maakt. Ook het hergebruiken van modules of artefacten in modellen wordt hierdoor bemoeilijkt. Neem bijvoorbeeld een onderzoeksgroep, die een model heeft ontwikkeld dat verkeerssituatie beschrijft. Een andere onderzoeksgroep doet gelijkaardig onderzoek en wil het model van de eerste onderzoeksgroep uitbreiden. Echter heeft deze onderzoeksgroep geen ervaring met

de tool die gebruikt is voor het originele model. Het vertalen van het model naar een andere tool, zodat de semantiek wordt behouden, is praktisch onmogelijk.

Wij trachten dit probleem te verhelpen door te onderzoeken hoe ABMS wordt gerealizeerd in tools en formalismen, en stellen een formalisme voor met precieze semantiek, dat de essentiele karakteristieken van ABMS implementeerd. Bovendien, presenteren we een tool die dit formalisme implementeert. Deze tool, SARLforSIM, evalueren we tegenover een selectie van reeds bestaande tools. Dit doen we aan de hand van een vergelijkende studie, waarin we eenzelfde model implementeren in elke tool en kijken we naar hoe deze tools verschillende concepten en technieken uit ABMS verwezenlijken. Ook onderzoeken we de reproduceerbaarheid van dit model in de verschillende tools.

CHAPTER 1

Introduction

Agent-based modelling and simulation (ABMS) is a powerful technique for modelling complex systems as a collection of autonomous, interacting, decision making entities [14, 52]. These entities or agents are able to observe and act within an environment and interact with other agents in the model. Agents have a behaviour, which describes how they will act and interact in certain situations. Their behaviours are influenced by their observations of the environment as well as the information they pass to each other. Fabien Michel [58] calls agent-based models virtual micro worlds or "microcosms", of which it is possible to control all characteristics and reproduce series of experiments as in a laboratory.

The field of ABMS, however, is divided. Multiple research groups from different domains (sociology, biology, economy, etc.) work on ABMS in a rather isolated way. This results in a state of the field where there is no universal agreement on the definition of an agent based model. Although these different groups seem to agree on concepts more than they disagree, categorizing a model as agent-based or not is no trivial task, as demonstrated by the collection of definitions proposed by Charles Macal [51]. This lack of uniformity can also be seen in the tools that realize ABMS. Tools such as NetLogo [79] and MASON [49] have their own interpretations of the semantics of an agent. Moreover, these tools do not define their semantics precisely, which complicates the analysis of models in such tool drastically, as well as the reusability of artefacts [41].

We will tackle this problem by investigating how agent-based modelling is currently realized and propose a formalism with precise semantics that captures the essential characteristics of agent-based modelling. Additionally, we present a tool which implements the proposed formalism. This new tool, SARLforSIM, is evaluated against

a selection of current agent-based modelling and simulation tools. We performed a comparative study by implementing the same population dynamics model in each of the tools and investigated how they realize specific the features of an agent-based model. Finally, we also investigated the reproducibility of the model in the different tools.

1.1 Research questions

To achieve to main research objective, we aim to answer to following central research question:

Is it feasible to define precise semantics that represent the essence of agent-based modeling and simulation, and implement them in a dedicated simulation platform?

To answer the central research question, we decomposed the main problem into components or sub-problems. We identified following sub-problems:

What are the key elements in agent-based formalisms? The first step of this research is to investigate the paradigm of agent-based modeling and simulation. Currently, the paradigm is used in a wide variety of research fields [51]. This resulted in many views and opinions on the topic. We will analyze what sets agent-based modeling apart from other modeling paradigms by identifying the key elements of agent-based modeling and simulation. The agent paradigm is not limited to agent-based modeling and simulation. We will also look into the related field of multi-agent systems.

How well are the concepts of ABM realized in current formalisms and platforms? Many languages, tools, and platforms have been developed that implement the agent-based modeling and simulation paradigm. We will analyze how these tools realize the key features of agent-based modeling and simulation.

Can we propose a precise semantics for ABM? A precise semantics provides as unambiguous description of the semantics of a formalism or platform. All platforms dedicated to agent-based modeling that we reviewed previous to this study, semantics were not defined precisely. We will propose a formal semantics of agent-based modeling and simulation, based on essential features that we identified previously.

Can we propose an m&s platform that correctly implements the proposed formal semantics? On top of proposing a formal semantics for agent-based modeling and simulation, we will show how to integrate this with a simulation platform.

1.2 Method

To structure our approach during this study, we followed the Design Science Research Methodology (DSRM) proposed by Peffers, Tuunanen, Rothenberger, and Chatterjee [67]. Though it was originally proposed for design science (DS) in information systems, we found the structured approach very useful for our application. Figure 1.1 shows an overview of the methodology.

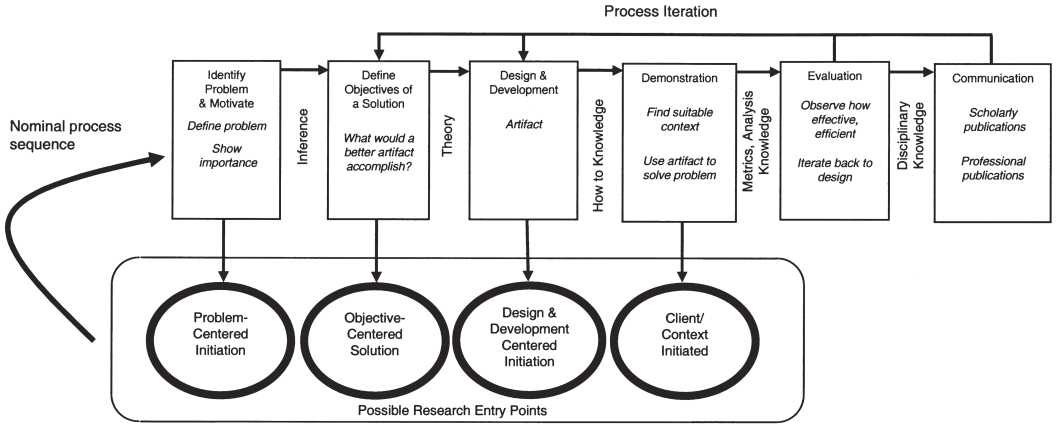


Figure 1.1: Overview of DSRM [67]

Based on the DSRM, we established the following phases in our research project:

1. **Problem identification and motivation**
   In the initial phase, we will identify research problem and motivate its importance. We will analyze existing tools as well as the paradigm of agent-based modelling. This knowledge will allow us to identify the variation points of definitions agent-based modelling and simulation and related concepts, as well as the variation points in the provided tooling.

2. **Define objectives**
   In the second phase, we will define the objectives for a good solution. These objectives will be distilled from the previous phase. These goals will objectively show the validity of our solution to the problem.

3. **Design and development**
   In the design and development, we will define an architectural design for the simulation platform. We will then implement said platform in order to perform experiments that will show whether our goals are achieved.

4. **Demonstration**
   We will demonstrate to MS experts within the MSDL research group successive prototypes of our tool and gather their feedback, which can then be incorporated in future versions of the framework.

5. **Evaluation**
   In the evaluation phase, we will evaluate the proposed platform with several methods. We will define a set of test cases where the output trace of the model in the tool is compared to the expected output trace of the model in the formalism. The second method is to implement a running example in our tool as well as in other tools. This will allow us to analyze in detail the benefits as well as the downsides and pitfalls of our tool. This also allows us to inspect reproducibility between all of these tools.
6. **Communication**
   This research is part of a master dissertation at the University of Antwerp [2] and will be publicly available at the university's library.

1.3 Organization of the document

This document is divided into multiple chapters. In chapter 2, we provides the necessary background on the concepts and ideas related to this work. Chapter 3 discusses the analysis we performed on the field of agent-based modelling and simulation. In chapter 4, we present our formalism with precise semantics, as well as the SARLforSIM tool, which implements our formalism. In chapter 5, we discuss the evaluation of our tool. Chapter 6 lists a collection of studies that relate to this thesis. Finally, chapter 7 concludes this thesis.

CHAPTER 2

Background

In this chapter, we will elaborate on background that is necessary to understand the rest of the document. We will first elaborate on modelling and simulation (M&S), where we discuss the different implementations of time and the concept of a simulation loop. We also discuss the underlying theory we use for language engineering, which is then applied to develop new modelling formalisms. We will also give an overview of agent-bases modelling, multi-agent systems, and agents. We conclude with a running example that demonstrates a typical application of agent-based modelling.

2.1 Language engineering

In this section we will elaborate the view on language engineering that is the basis for our research. This discussion is based on the discussion in [84].

A formalism, is defined by its syntax as well as its semantics, as seen in Figure 2.1. Syntax is further split into concrete syntax and abstract syntax. Semantics, in turn, is defined by the semantics mapping and the semantic domain. We will now define each of these concepts in more detail:

- **Abstract syntax:** The abstract syntax defines all constructs in the formalism and how they can be combined.
- **Concrete syntax:** The concrete syntax defines the visual representation of the constructs. These can either be graphical or textual. Also, multiple concrete syntaxes can be defined for a single abstract syntax.

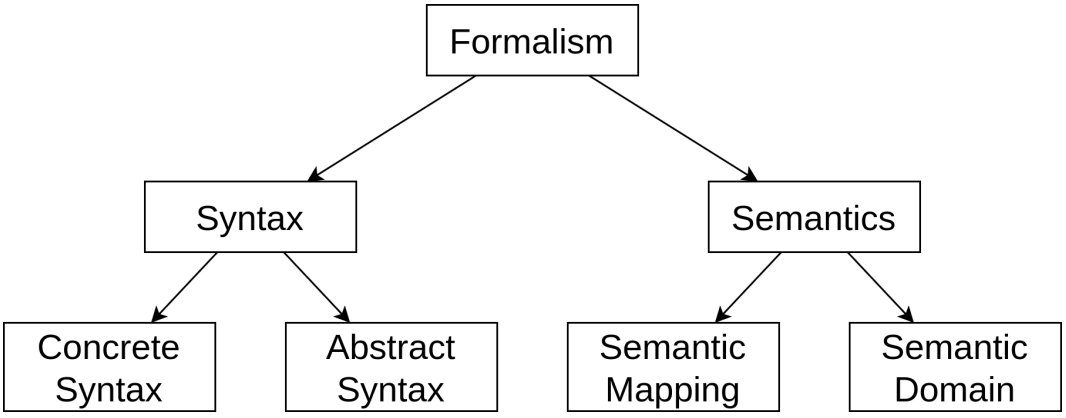


Figure 2.1: The different aspects of a formalism [84]

- **Semantics:** The semantics define the meaning of the models created in the formalism. The semantics comprises both the semantic mapping, which translates a valid instance in the formalism to an instance in the semantic domain. The semantic domain is a formalism with well-known semantics.

Generally speaking there are two types of semantic mapping: translational semantics and operational semantics. In translational semantics, a model in the formalism is translated to a model in another formalism with known semantics. The two models are equivalent with respect to the properties under study. In operational semantics, the mapping will actually execute or simulate the mapped model. Here, the semantic domain is the collection of output traces.

2.2 Modelling and simulation

In this section, we elaborate on modelling and simulation concepts that we use throughout this document.

2.2.1 Modelling paradigms

Throughout this document, we will refer to agent-based modelling and simulation as a modelling paradigm. Our interpretation of a paradigm is that it is a collection of ideas, methods, and approaches on how to solve a certain problem. For example the object-oriented programming paradigm comprises a set of ideas and methods to solve problems by dividing the problem into modular components, called objects. A paradigm does not impose a precise set of rules, but rather represents a general idea that can be applied in various ways.

A modelling (and simulation) paradigm is a paradigm that relates to modelling and simulation. The agent-based modelling paradigm advocates how complex systems

can be modelled by interacting, individual entities. Therefore, we consider it a modelling paradigm.

A paradigm is often accompanied by a set of tools that implement that paradigm. This means that the tool is based on the ideas presented in the paradigm and allows users to solve problems more easily by capturing concepts from the paradigm as first-class abstractions and providing structured methods to solve the problem based on the paradigm.

2.2.2 Virtual time

In a modelling and simulation context, a time base is an abstract way of ordering observations made on a system [92]. This abstract ordering of events that model the flow of actual time is often referred to as logical time. In contrast, in real world systems, such as MAS, time is measured by an actual clock. Such a time base is called physical time. In this section we will discuss the three possibilities of modelling time (discrete-time, discrete-event, continuous time).

Discrete-Time

The first model of time we discuss is discrete-time models. These types of models assume that the state changes on discrete points in time, which are typically equidistant. At each point or time-step, the model is in a certain state and defines how it state changes. Though discrete time models are especially suited for modelling systems that exhibit step-wise behaviour, it is also frequently used to approximate continuous systems.

To give a formal representation of how the state of the model changes over time, we define the state at time-step i as s_i. The input from the environment of the model is defined as x_i. If input of the environment is generated between two time steps, the input is buffered until the next time step is simulated. Then the next state of the model is given by $\delta(q_i, x_i)$ where δ is the state transition function. The output that is generated, y_i, is given by the output function $\lambda(q_i, s_i)$. Or in another representation:

$$q_{i+1} = \delta(q_i, x_i)$$

$$y_i = \lambda(q_i, x_i)$$

In Figure 2.2, a graph is shown depicting how state changes over time in a discrete time model. Note that in between time steps, the state of the model is undefined.

Discrete-Event

In discrete time models, at every time step each component performs a state transition, even if the actual state of the component does not change. Often, not every

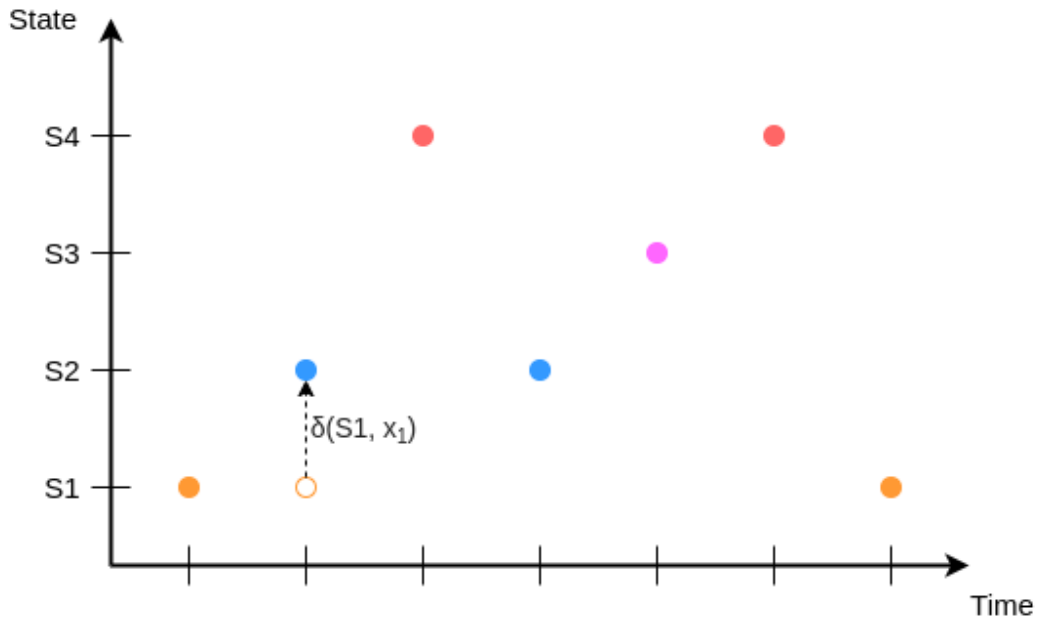


Figure 2.2: State changes over time in a discrete time model

component wants to execute a state transition at every time step. The discrete event paradigm aims to solve this problem by allowing each component to schedule its own state transitions at arbitrary time steps. The possible states of the model are given by a discrete set. State updates can occur at arbitrary points in time and time is considered continuous. This means that, unlike discrete time models, the state in between two time steps is defined, as seen in Figure 2.3. This allows input from the environment at any point in time.

Continuous time models

In discrete time models as well as discrete event models, the state of the model can only be changed at certain points in time. In the modelling approach of continuous time modelling, this done quite differently. Here we do not specify state transitions, but rather the rate of change of state variables. Given the initial state of the model as well as the rate of changes of each value in that state, we can compute the state at an arbitrary point in time.

To summarize, continuous models have a continuous representation of time as well as a state that changes continuously through time.

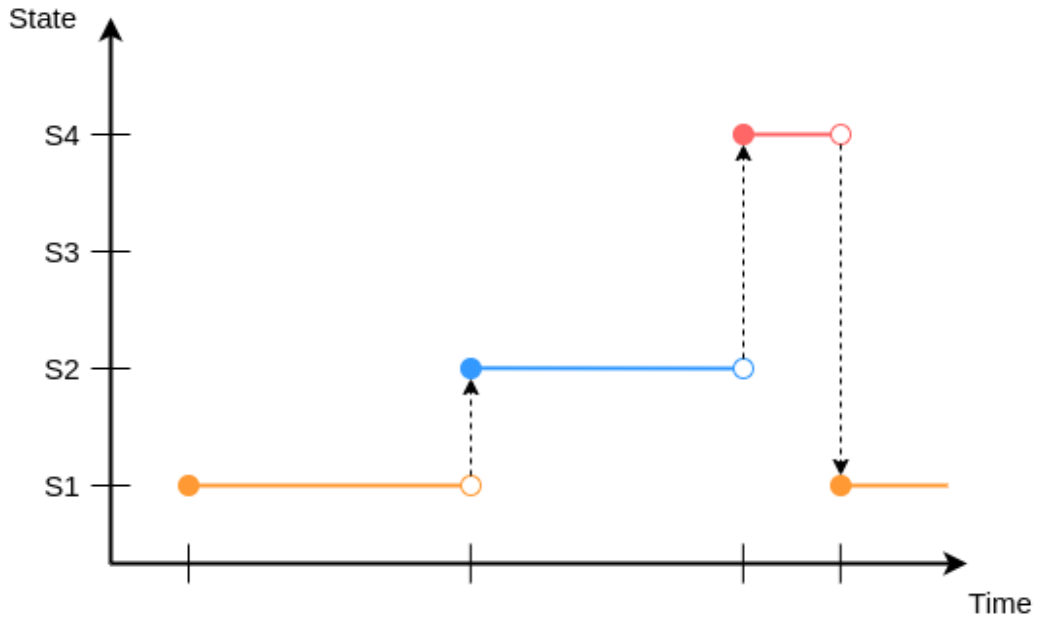


Figure 2.3: State transitions in a discrete event model

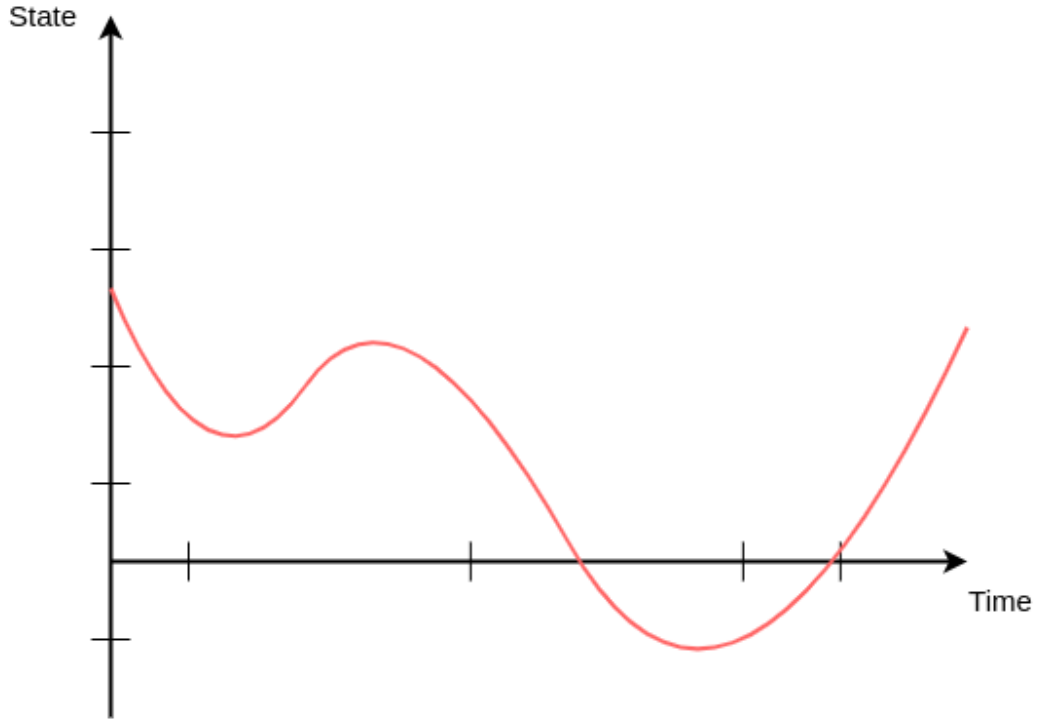


Figure 2.4: State in a continuous time model

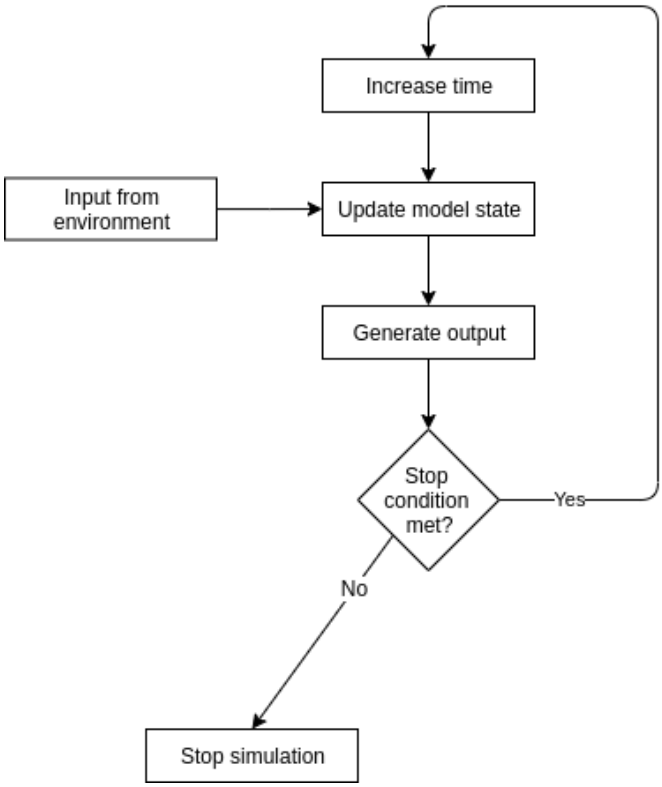


Figure 2.5: A flow chart of the simulation loop

2.2.3 The simulation loop

In the Section 2.1, we mentioned the operational semantics of a formalism. This type of semantic mapping will simulate a valid sentence of a formalism. Within the context of modelling and simulation, this is done by a simulator. In this section, we will discuss the simulation process in more detail as well as how time is represented in modelling and simulation contexts.

To simulate a model, simulators make use of a simulation loop. Each iteration of this loop updates the virtual time and simulates an update of the state of the model according to the specification of the model. During each iteration of the simulation loop, we can output information about changes of the state. This generates an output trace, which can then be inspected to find the cause of certain behaviours of the model, for example.

This loop is executed repeatedly until a stop condition is met. Fig 2.5 shows a flowchart of the the simulation loop.

2.3 Agent-based modelling

The main topic in this thesis is agent-based modelling. This modelling and simulation paradigm advocates to model all individual entities in a system as agents. ABM exploits a bottom-up approach when modelling a system, instead of a top-down approach used in for example system dynamics [26] and causal block diagrams [37]. In a top-down approach, the system is first modelled as a whole. Then the model can be refined to make conclusions about the effects of various influences. In a bottom up approach, the system is modelled by explicitly modelling each individual aspect of the system. Then we inspect how different configurations of these aspects influence the global state. In this section, we provide an overview of the history of agent-based modelling as well as a review of definitions found in the literature. This section is copied from our preparatory literature review [47].

History of agent-based modelling

In [58], Fabien Michel clearly describes the history of agent-based modelling and simulation. In this section we will give a summary and add some extra information we found in literature.

"Modelling of complex systems has always been a motivation for researchers" [58]. A historical example of such a model is the predator prey model, originally proposed by Volterra [87]. This continuous deterministic model captured the population dynamics of multiple species living in the same habitat with differential equations. This model produced intuitively sound results and therefore became very popular. In a predator prey situation, it displayed an oscillation in both populations, as seen in Figure 2.6. When the prey population is high, the predator population would grow, resulting in a decrease in the prey population. Contrariwise, when the the prey population is low, the predator population would decrease, resulting in a growth of the prey population.

Results from experiments on real test predator/prey habitats, however, showed non-

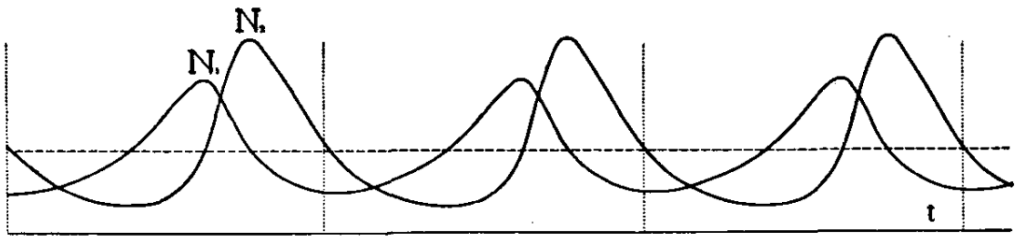


Figure 2.6: The oscillation of the populations N_1 and N_2 in the predator prey model [88]

oscillating behaviour. This caused the need for more complex models.

"With the advent of computer science, the possibility of simulating stochastic models efficiently presented itself" [58, 10]. Stochastic models allowed for incorporating the inherent non-deterministic behaviour of real life interaction situations. However, these models exhibit much more complex behaviour and a high variability even used with the same parameters.

All of these approaches still face a set of problems [33]:

- Only a global perspective is possible
- Equation parameters hardly take into account the complexity of micro-level interactions
- The modelling of individual agents is impossible
- Integrating qualitative aspects is hard

The first approach to deal with these problems is called micro-simulation and is originally suggested by Orcutt [64]. The basic principle of micro-simulation is to model the micro-level behaviour of a system under study with rules that apply to attributes of micro-level entities, resulting in a change of state or behaviour of those entities. The incorporation of the micro-level into the development of models, allowed the generation of better and more informative results for social system dynamics. *"Though this modelling paradigm originated in the field of social sciences, it can be seen as a forerunner of individual based modelling."* [58]

Agent-based modelling takes the integration of the micro-level a step further. Apart from modelling entities and their behaviour, ABM suggests to model their interactions with each other and actions on the environment explicitly. The popularity of ABM can be credited to a number of factors. First, ABM is very flexible. Models can easily be updated by adding new types of agents or changing the populations of agents. ABMs also allow fine-grained control over the simulation. In contrast to other modelling techniques, ABM does not abstract individual behaviour and interactions. Being societies by themselves, and being built on the same basis as any complex systems, agent based models prove to be "artificial micro-worlds", of which it is possible to control all characteristics and reproduce series of experiments as in a laboratory [58].

Definition review

In the current literature, we found several definition for agent-based models. We will now list the most prominent and discuss them.

Due to the wide variety of applications of ABM, it is not trivial to give a uniform definition. Charles M. Macal therefore proposed a collection of four definitions for different types of agent based models [51]. The categories identified by Macal are

Individual Agent Based Models, Autonomous Agent Based Models, Interactive Agent Based Models, and *Adaptive Agent Based Models.*

The first category are defined as follows:

Definition 1 *An individual ABMS is one in which the agents in the model are represented individually and have diverse characteristics.*

In these models, agents are hardly autonomous. Each agent does not perceive any information of its environment and thus can not take any decisions based on those percepts. Their behaviour is described in a script and no individual state is necessary. Note that the agent's behaviour in these models is even more simplistic than the simple reflex agent. These models aim to reflect the individual behaviour of each agent and the model is constructed in a "bottom-up" approach that is characteristic for agent based modelling.

The second category of agent-based models is defined as:

Definition 2 *An autonomous ABMS is one in which the individual agents have internal behaviours that allow them to be autonomous, able to sense whatever condition occurs within the model at any time, and to act on the appropriate behaviour in response.*

Autonomy is widely considered a fundamental property of agents. This definition also points out that agents are situated in an environment. This property is strongly connected with autonomy, since an agent needs to perceive some external data to decide which action to take. The environment provides this data.

Interactive agent-based models are defined as:

Definition 3 *An interactive ABMS is one in which autonomous agents interact with other agents and with the environment.*

This category introduces an important feature of multi-agent systems, namely interaction between agents. The interaction in these models is either direct, indirect, or both. The agents in the previous category are only able to perceive their environment, but are unable to act upon it. This prohibits indirect communication.

The last category is defined as:

Definition 4 *An adaptive ABMS is one in which the interacting, autonomous agents change their behaviours during the simulation, as agents learn, encounter novel situations, or as populations adjust their composition to include larger proportions of agents who have successfully adapted.*

Here agents are able to change their behaviour during the simulation.

A completely different categorization is given by E. Bonabeau [14]. Bonabeau does not categorize agent based models on the capabilities of the agents, but rather the

application of the models. He presents four categories: Flow Models, Market Models, Organization Models, and Diffusion Models.

Flow models are characterized by featuring a large amount of mobile agents in a particular situation. The interest of the modellers is how the agents move through their environment and create certain movement flows. Flow models can be further specialized into evacuation models and flow management models. Evacuation models aim to analyze situations were agents are trying to exit a room with limited exits, while the agents exhibit irrational herding behaviour and obsessive personal interests. Flow management models model environments with mobile decisive agents.

Market models aim to model the dynamics of stock markets and auctions. Agent-based models are preferred approach over spreadsheet models or system dynamics, because they do not achieve the same deep insights. The behaviour of the market emerges out of the interactions of the players, who change their behaviour when the market changes.

Organization models are used to analyze emergent collective behaviour of an organization. An example of this is operational risk. Operational risk factors are often largely internal to the organization and clear mathematical or statistical link and the size and frequency of operational loss does not exist.

Diffusion models are applied to cases where people are influenced by their social contexts. Agents-based models improve on classic models in that they can model locality. In these models, agents only communicate with their neighbours.

There is no universal agreement on the definition for ABM [52]. This is apparent from the several definitions for different types of agent-based model by C. Macal. He provides four definitions for agent-based models based on the internal complexity of the agents of the model. There is big lack of unification in the field, however, we take the view that it is possible to provide a clear all encompassing definition for ABM. Therefore, we did not focus on the complexity of a model, but rather on what makes ABM stand out as a modelling paradigm.

Definition 5 *An agent based model is a model in which:*

- *Each individual of a system and its behaviour is modelled (bottom-up), rather than the global behaviour of the system (top-down).*
- *Each of the entities is modelled as an agent, where the interactions between agents and actions on the environment are modelled explicitly.*
- *The environment in which the agents are situated is modelled explicitly.*

Comparison with other modelling paradigms

Agent-based modelling is a popular approach for modelling and simulation [51, 58]. In this section, we will discuss what sets this paradigm apart from other modelling

paradigms.

In agent-based models, individual entities are not abstracted. Rather, they are the key elements in the models. This is in stark contrast with equation-based modelling (EBM), which abstracts away individual behaviour and aims to capture global system behaviour with a set of differential equations. Examples of EBM approaches are causal block diagrams [37], and system dynamics [26].

In essence, equation-based models provides an approach to *"numerically characterize the evolution of a system from its parameters"* [58], while ABM allows the modeller to create a habitat of agents which can interact with each other and their environment, and explicitly model their individual behaviour. This creates some kind of 'virtual micro-world', which can be analyzed as a small version of the real system.

Though ABM is usually compared to EBM, it also serves as an alternative to stochastic modelling [69]. In stochastic modelling, a system is described with probability distributions and information about the system is inferred through probabilistic reasoning. This modelling paradigm aims to model variability and randomness that is often found in real-world systems.

In agent-based modelling, stochasticity is modelled in the behaviour. Instead of inferring information through probabilistic reasoning, the system is simulated and information can be retrieved by analyzing the interactions of the agents.

2.4 Multi-agent systems

Besides agent-based modelling and simulation, the field of multi-agent systems features the concepts of agents and environments extensively. In this section, we provide a brief overview of this domain, which we duplicated from our preparatory literature study [47].

History of multi-agent systems

The first steps into multi-agent research began in 1980, when a group of AI researchers organized the first workshop for distributed artificial intelligence at MIT. This new field of study concerned itself with issues of how intelligent problem solvers could coordinate effectively to solve problems [42]. Initially the field was divided into two camps[15]: Distributed Problem Solving (dps) and Multi-Agent Systems (mas). However, recently the term multi-agent systems has become a more general term, encompassing all distributed problem solvers.

The first model for distributed problem solvers were actors [6, 4]. Actors are self-contained components in a distributed system that can make autonomous decisions. The primitives of an actor are:

- Create an actor
- Send a message
- Change state

Actors are blind to their environment. They only receive messages from each other and decide how to respond to these messages. Though agents and actors are two distinct concepts, they share technical and historical connection [44] and actors have been defined as "computational agents" in the past.

Actors were the first implementation format for multi-agent systems. Now many architectures and methodologies have been defined that are specifically developed for multi-agent systems. The PASSI methodology [25] provides a requirements-to-code methodology for developing agent systems from both the OOP perspective as well as the AI perspective. The ADELFE methodology [12, 68] was specifically developed for designing adaptive multi-agent systems [21]. In these systems agents learn the topology through cooperative and non-cooperative states. Next to design methodologies, specific architectures have been developed for implementing distributed agents systems. Some examples are the open agent architecture [22], and Cougaar [40]. Also, dedicated tools were developed that allow users to develop multi-agent systems more easily by providing domain specific abstractions, such as JADE [11].

Some early applications of MAS include *air traffic control* [20], the distributed vehicle monitoring task [29], and blackboards [24]. Currently, MAS has a wide variety of applications ranging from cloud computing to robotics and even city and built environments [28].

Definition review

In this section we will review some of the definitions of multi-agent systems we found in the literature.

The first definition comes from a survey by P. Stone. [77].

Definition 6 *A multi-agent system is 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).*

Stone focuses on the distributed and collaborative aspect of multi-agent systems. The global goal of the system is divided into smaller parts and given to agents. To achieve the global goal agents need to collaborate.

The second definition we reviewed, we derived from *An Introduction to Multi-Agent Systems* [33].

Definition 7 *A Multi-Agent System (MAS) is an extension of the agent technology where a group of loosely connected autonomous agents act in an environment to achieve a common goal. This is done either by cooperating or competing, sharing or not sharing knowledge with each other.*

Ferber et al. note that agents in a multi-agent system are not necessarily collaborative. In some cases to achieve a certain goal it is necessary for agents to compete against each other.

In the book *Multi-agent systems: a modern approach to distributed artificial intelligence* [90] the major characteristics of a multi-agent system are stated.

Definition 8 *The major characteristics of a multi-agent system are:*

- *Each agent has just incomplete information and is restricted in its capabilities.*
- *System control is distributed.*
- *Data is decentralized.*
- *Computation is asynchronous.*

This definition again highlights the distributed nature of multi-agent systems. It also discusses the more technical details of multi-agent systems.

In Table 2.1, we created a table with important keywords and their occurrence in the reviewed definitions. We clearly see that all definitions are very similar. We can conclude that essential features of a multi-agent system are:

- A MAS is a distributed system with agents as central entities
- Agents are limited in their knowledge and capabilities
- Agents need to collaborate to achieve their goals

Table 2.1: Table of occurences of concepts in the reviewed definitions

	P. Stone [77]	Ferber et al. [33]	G. Weiss [90]
Agents	✓	✓	✓
Limited knowledge	✓		✓
Limited capabilities	✓		✓
Distributed system	✓	✓	✓
Common goal		✓	
Collaboration	✓	✓	
Competition		✓	
Asynchronous computation			✓
Decentralized data			✓

Single-Agent Systems

In addition to distributed multi-agent systems, there are centralized single-agent systems [77]. These systems, which were popularized by the book *Logical foundations of Artificial Intelligence* [34], consist of a single agent in an environment. These systems obviously omit the need for communication. In this contexts, the term agent refers to a distinguishable entity that is able to sense and act on its environment deliberately, Figure 2.7.

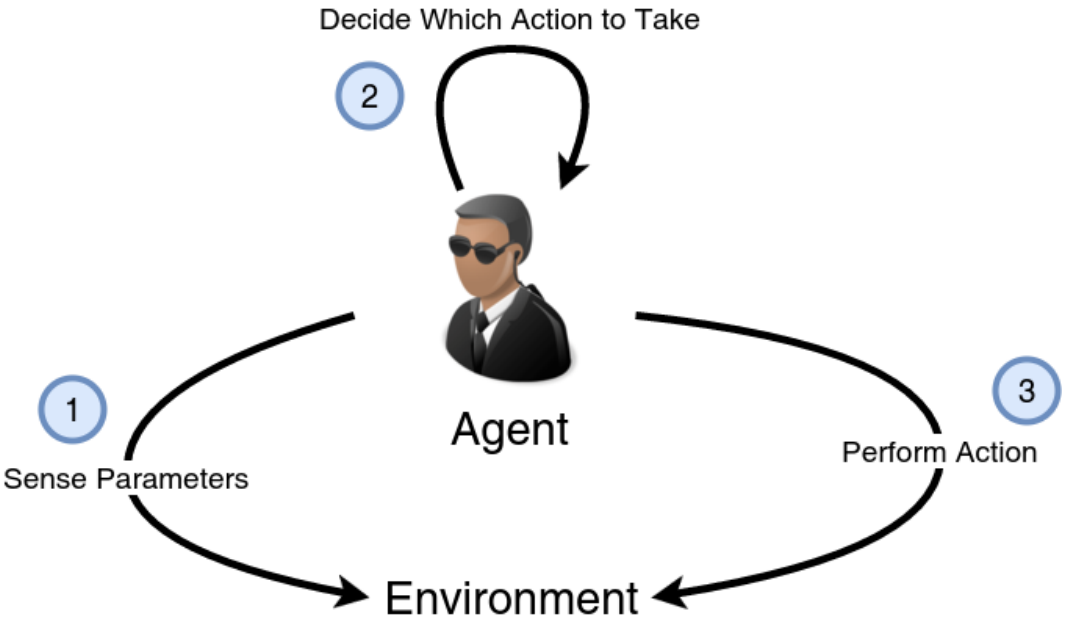


Figure 2.7: Typical workflow of an agent

According to [77], a single agent system can consist out of multiple entities. However, these entities act as actuators for a central agent (or central process) and the agent will identify each of these separate components as part of itself.

In [77], another type of single-agent systems is described. In these systems, multiple agents are present. However, an agent has no representation of the other agents present. The other agents are considered to be part of the environment and can not be contacted through specialized agent-to-agent communication protocols.

2.5 Agents

The agent is the central concept in both multi-agent systems as well as agent-based modelling [28, 51]. In this section we provide an overview of this concept as well as a review of definitions found in literatured. This section was copied from our prepatory literature study [47].

History of agents

There are three fields of research that contributed to the concept of agent [42]: Artificial intelligence [74], object-oriented programming [16] and concurrent object based systems [6, 5], and human computer interface design [53].

Artificial intelligence is undoubtedly the main contributor to the field. Ultimately, AI is all about building intelligent artefacts, and if these artefacts sense and act in some environment autonomously, then they can be considered agents. This makes agents the central study in artificial intelligence [74]. However, up until the 1980's, little effort was put in the research of intelligent agents. In this time period, research was focused more on the individual components of an agents, such as question-answering systems, theorem-provers, vision systems, etc. In 1987, Genesereth and Nilsson published an influential paper [34] which caused the concept of the whole-agent to be widely accepted in the field of artificial intelligence.

Agents that exhibit simple reactive behaviour were the primary model adapted by psychological behaviourists such as Skinner [76]. However, most AI researchers deem these agents to be too simple to provide much leverage. Rosenshein [73] and Brooks [18] question this assumption. Currently a lot of research is focused on finding efficient algorithms for keeping track of complex systems. An impressive example is the Remote Agent Program that controlled the Deep Space One spacecraft [61].

A more complex behavioural model was developed that introduced goals. Goal-based agents did not simply react to perceptions in their environment, rather they decided on an action to perform that optimized conditions to reach a certain desired end-state or goal. With goal-based agents, the application of agents in robotics was explored. The first robotic implementation of a logical, goal-based agent was Shakey the Robot [62]. The agent approach also gained a lot of attention in the field of software engineering [91]. Shoham [75] developed a new programming paradigm based on agents, agent-oriented programming. The goal-based view of agents also dominates the field of distributed artificial intelligence, where a problem is divided over multiple solvers, the agents, in a multi-agent system [42].

In 1987, the goal-based model was further specialized into belief-desire-intention agents [17]. Goals were further specified into desires (general goals) and intentions (currently pursued goals). BDI agents also have beliefs of their environment. This allows agents to remember information from their environment that they can't perceive.

Research has also been devoted to adding learning capabilities to agents [19, 59]. Learning agents can be divided into two groups. Agents that can adapt their behaviour by studying the result of their actions and agents that can adapt there social interactions to become more competent.

In recent years, the interest in agents and agent design has increased rapidly. This can be partly credited to the growth of the internet, which provides a massive

collection of data that can be used as environment for agents [30, 45]. Also the rise of cyber-physical systems is a growing field that utilizes the concept of agents for automation of development in industry 4.0 settings [46, 89, 48].

Definition review

In this section, we will review a set of definitions for the concept of agent. Since an agent is a concept in MAS as well as ABM, we reviewed definitions in both fields to see whether they have different conceptions about the term. We focused on surveys to filter out definitions that are biased to a specific field of research.

The first definition is derived from a survey of Nick Jennings and Michael Woolridge from 1996 about software agents in multi-agent systems [43].

Definition 9 *Agents should have following key hallmarks:*

- *Autonomy: Agents should be able to perform the majority of their problem solving tasks without the direct intervention of humans or other agents, and they should have a degree of control over their own actions and their own internal state.*
- *Social ability: Agents should be able to interact, when they deem appropriate, with other software agents and humans in order to complete their own problem solving and to help others with their activities where appropriate.*
- *Responsiveness: Agents should perceive their environment (which may be the physical world, a user, a collection of agents, the INTERNET, etc.) and respond in a timely fashion to changes which occur in it.*
- *Pro-activeness: Agents should not simply act in response to their environment, they should be able to exhibit opportunistic, goal-directed behaviour and take the initiative where appropriate.*

Jennings et al. identify agents to be autonomous communicative entities. They hold a degree of autonomy and should be able to solve a majority of problems by themselves. They should be able to communicate with each other on their own initiative.

Jennings et al. also introduce some terms, such as environment and goals. An agent's environment provides the conditions under which an entity (agent or object) exists [63]. Agents perceive information from their environment and can act upon it, which can in turn cause a change in the environment. Agents also have a local goal. This is a set of conditions that the agent tries to satisfy as best as possible.

The second definition is from Charles M. Macal and Michael J. North. They provide a definition for agents from the perspective of agent-based modelling in *A Tutorial on Agent-Based Modelling*[52].

Definition 10 *An agents has the following essential features:*

- *An agent is a self-contained, modular, and uniquely identifiable individual.*
- *An agent is autonomous and self-directed*
- *An agent has a state that varies over time*
- *An agent is social having dynamic interactions with other agents that influence its behaviour.*

Both definitions are from a different perspective, however, some features recur in both definitions. Both definition identify that agents should by autonomous and communicative. Macal and North's definition, does not mention that agents are situated in an environment or have goals.

A paper that was included in *Multi-Agent systems: Simulation and applications* [83] by Fabien Michel et al. also provides a definition from the modelling and simulation point of view [58].

Definition 11 *An agent is a software or hardware entity (a process) situated in a virtual or a real environment:*

1. *Which is capable of acting in an environments*
2. *Which is driven by a set of tendencies (individual objectives, goals, drives, satisfaction/survival function)*
3. *Which possesses resources of its own*
4. *Which has only a partial representation of this environment*
5. *Which can directly or indirectly communicate with other agents*
6. *Which may be able to reproduce itself*
7. *Whose autonomous behaviour is the consequence of its perceptions, representations and interactions with the world and other agents*

This definition mentions that agents can have a partial view of its environment. This refers to locality of an agent. An agent can only perceive information in its surroundings. E.g. a person can only see what is happening in the room he is in, not a neighbouring room. He can however remember what the room looked like when he left. When an agent only has a partial perception of his environment, the internal

representation of an agent's environment is often referred to as the agents belief [36].

In a recent survey from Ali Dorri et al. [28] about MAS, the authors proposed a general application-independent definition for an agent.

Definition 12 *An agent is an **entity** which is placed in an **environment** and **senses** different parameters that are used to make a **decision** based on the **goal** of the entity. The entity performs the necessary **action** on the environment based on this decision.*

Though not mentioned explicitly, the definition states that agents should be autonomous. By sensing the environment and choosing an action accordingly, the environment can not control the agents action directly.

To compare these definitions, we constructed Table 2.2, in which we listed important keywords from the definitions and put a check mark if the keyword is discussed in the definition.

We identified autonomy as the most important feature of agent-hood. Every definition that is reviewed mentions this feature explicitly or implicitly. Communication, environment, and goals are all mentioned in 3 out of the 4 reviewed definitions, making them also essential features for agent-hood. Pro-activeness is only mentioned explicitly in the definition of Jennings and Woolridge, however, pro-activeness is tightly coupled to goals. If an agent does not have the notion of goals, it can not exhibit opportunistic behaviour.

We also noticed that the definitions of Jennings and Woolridge and the definitions of Michel et al. are very extensive, while the definitions of Macal et al. and Dorri et al. are rather limited.

In the definitions of agents we did not notice any bias towards MAS or ABM. This makes agents a generic concept.

2.6 Running example

To explain concepts and theories throughout this document we will make use of an example that shows all aspects of agent based modelling. The purpose of this example is to increase the understanding of the reader as well as show the usefulness of our contribution. To increase the credibility of the example, we based our model on a real world application.

Table 2.2: Table of occurences of concepts in the reviewed definitions

	Jennings et al. [43]	Macal et al. [52]	Michel et al. [58]	Dorri et al. [28]
Autonomy	✓	✓	✓	✓
Communication	✓	✓	✓	
Environment	✓		✓	✓
Pro-Activeness	✓			
Beliefs			✓	
Goals	✓		✓	✓
Statefull		✓		
Self Reproducibility			✓	
Resources			✓	

2.6.1 Model requirements

Not any agent-based model is suited as a demonstrative example. It needs to show all aspects of agent-based modelling. To help us decide on a suitable example, we list all requirements in this section.

With the example, we want to show that our precise semantics implemented in our tool are the most appropriately way to implement agent-based models. Therefore we need to stave our decisions regarding how time is implemented, how dynamic structure is implemented, and that our formalism adequately represents the key features of agent-based modelling.

Requirements for the agents:

- **Reactive behaviour**: Agents need to be able to respond to events happening in the environment.
- **Proactive behaviour**: When no event occurs, agents will still schedule actions to seek out opportunities for the future.
- **Complex deliberations**: Agents need to come across situations where they need to deliberate their next action. This will show the autonomy of agents.
- **Communication**: Agents need to communicate with each other. Communication is a key aspect of collaborating agents.
- **Changing population**: To show the dynamic structure of an agent-based model, we need to create new agents during the simulation as well as let some of them die.

Requirements for the environment:

- **Dynamic environment**: The environment needs to change its state over time, even without agent interference.

- **Non-deterministic environment**: Agents should not be able to determine the result of their actions.
- **Partial view**: When agents perceive their environment, they can only perceive a part of it.

2.6.2 Model specification

For our example model, we propose a variation of a population dynamics model, originally proposed by Volterra [88]. Fabien Michel [58] argues that the agent-based modelling paradigm is most suited for modelling the predator-prey model for several reasons:

- It offers an individual perspective as well as a global perspective.
- It takes into account the complexity of micro-level interactions.
- It allows the modelling of individual actions.

For the purpose of the model, we will only model a prey species which is situated in an environment with limited resources. The resources, the food of the species, will be refilled periodically, while the agents have to seek opportunities to find enough food and to reproduce.

We will start by defining the environment of the model. The environment is divided into patches. Each patch contains an amount of grass that can be eaten by the species (goats). The amount of grass present in the patches depends on the season. Each season change, the amount of grass in each patch will be refilled. How much grass is refilled is randomly determined, but in summer it will stochastically be more than in winter. The season is also part of the state of the environment and changes periodically with a fixed interval.

The first type of agents that are present in the model are the sheep or prey agents. Each sheep maintains a state regarding its energy level, which decreases at a constant rate. Each day, the goat will observe its environment. When there is still grass on his patch, he will try to eat and refill his energy. Otherwise, the goat will migrate to an adjacent patch and starts grazing there. When a goat migrates, he takes multiple factors into account. The first one is the amount of grass in the neighbouring patch; the second one is the presence of a suitable mate; lastly, the amount of sheep that is already on the patch. No two male goats can stay on the same patch. If two or more male goats are on the same patch, they will fight to see who can stay and who should leave. Female goats will get pregnant if they are on a patch with a male goat. Then, after a gestation period, the goat will give birth to a new goat. If a goat runs out of energy, it will die. Otherwise, there age is randomly defined between 5400 days and 6480 days.

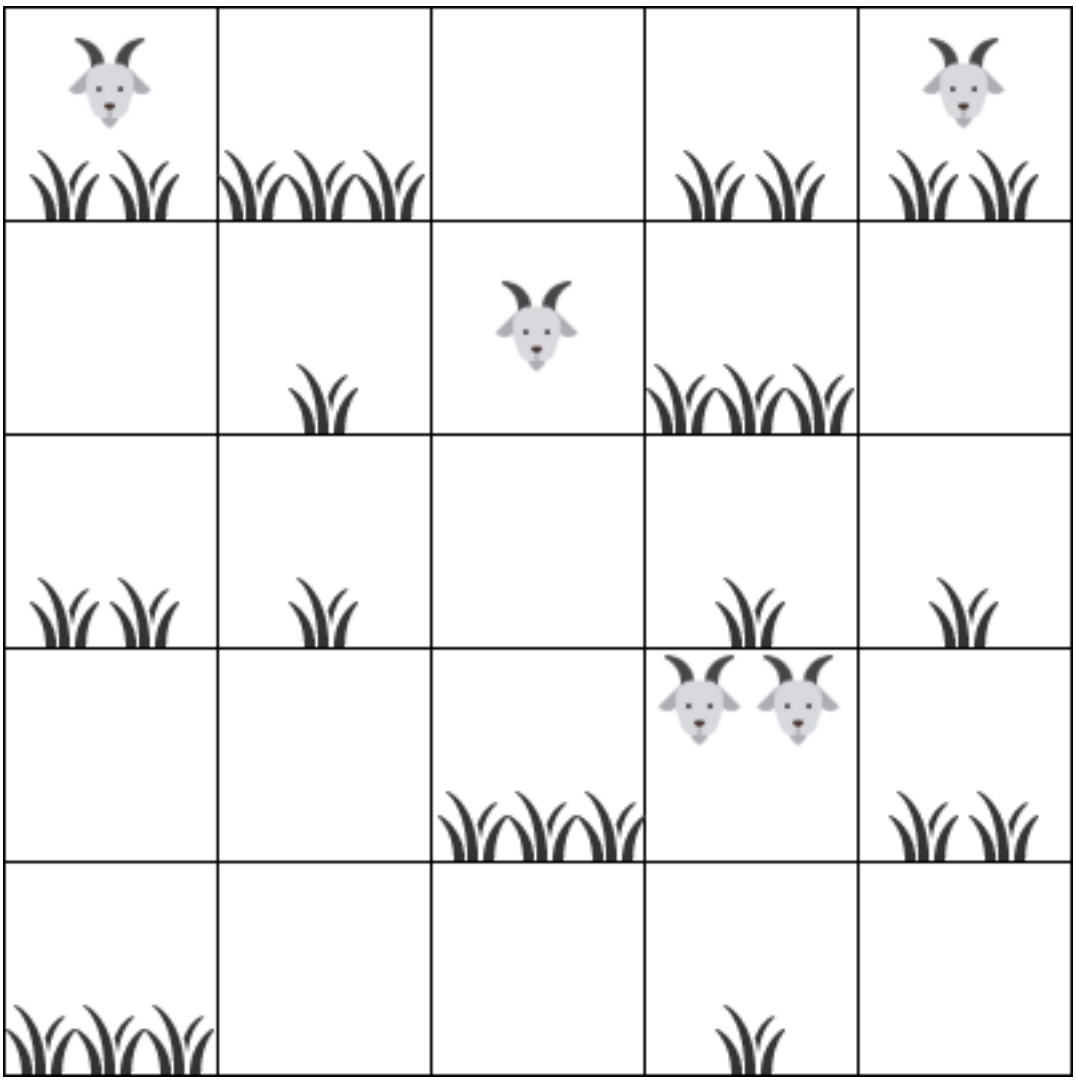


Figure 2.8: The running example

CHAPTER 3

Analysis of Agent-Based Modeling

The first step in our research process is the analysis of the agent-based modeling and simulation paradigm. In this chapter, we will identify the key features of an agent-based model, as well as discuss how they are realized in agent-based modeling and simulation tools. We will also analyze existing formalism and formalizations.

3.1 Key features of agent-based models

By studying current literature, we identified the key elements of an agent-based model. This discussion is based on the results of our literature study [47]. Macal and North [50] identified that an agent based model consists out of:

- A set of agents
- The relations of agents and methods of interactions
- The environment in which the agents are situated

In this section, we will discuss each of these components as well as how they are scheduled in a simulation environment.

3.1.1 Agents

The term agent originates in the field of Artificial intelligence [74] and is used to denote a individual, autonomous, problem-solving entity. The concept was later popularized in the field of multi-agent systems, which concerns itself with solving

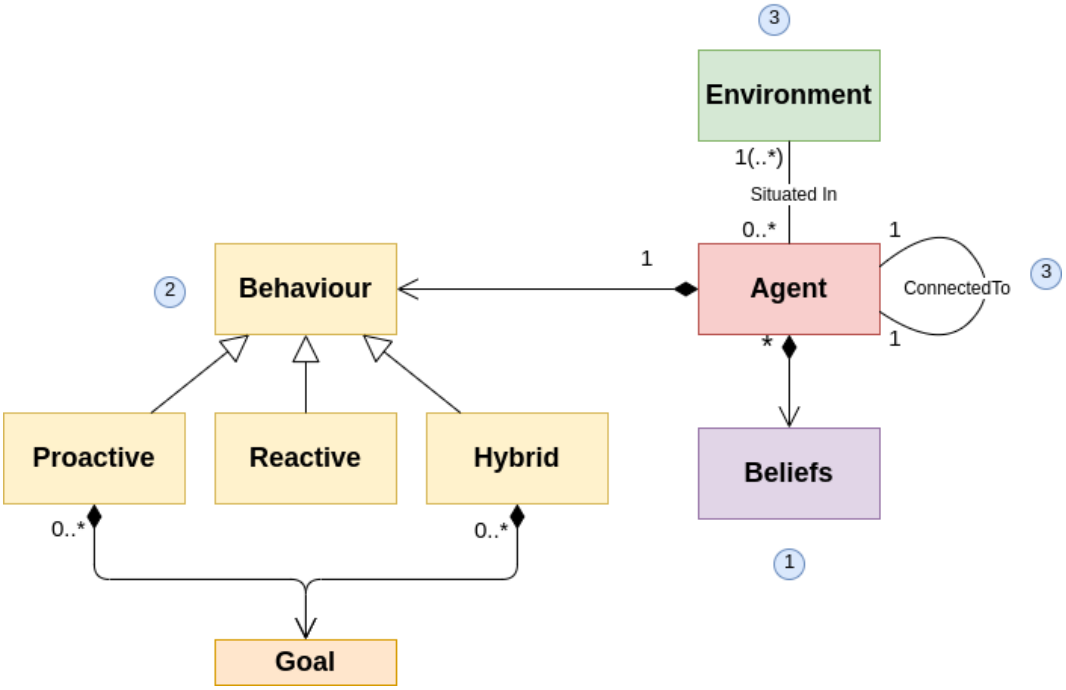


Figure 3.1: Diagram of an agent

complex problems by distributing the problem over multiple, collaborating agents [42]. In modelling and simulation, the need arose to model the individual entities of a system [58]. The concept of agents was adopted from multi-agent systems and agents were used to represent individual entities in a system under study.

In Figure 3.1, we show a diagram of the essential features of an agent. By inspecting current literature, we identified beliefs (1), autonomous behaviour (2), environment (3), and communication (4) as the essential features of an agent. We will now list each of the features with a more detailed description.

1) Beliefs The beliefs of an agent represent the agent's knowledge about itself, its environment, and other agents [36]. The beliefs of an agent make up its internal state and can be updated by perceiving their environment or by performing a certain action.

2) Behaviour The behaviour of an agent describes how an agent acts in certain situations. The behaviour is usually comprised of a set of rules that, based on the current state and perceptions, decide what action to perform. What sets agents apart from objects, is the constraint that an agent's behaviour should be autonomous. The chosen action can not be directly influenced by an external entity.

An agent's behaviour can either be reactive of pro-active (or a combination of both).

Reactive behaviour comprises a set of if-then rules, which define how an agent will react to a certain event in its environment, considering its internal state [86]. Proactive behaviour defines how the agent will act when no external event is taking place. This allows agents to seek out opportunities by internally scheduling actions [43]. Often, proactive behaviour is associated with goals. These goals determine which opportunities the agent will seek out.

A third type of behaviour is cognitive behaviour. This term is often used in literature to denote a behaviour which is a hybrid of proactive and reactive behaviour which takes into account both the beliefs of the agent, as well as certain goals the agent wants to achieve [86]. It is not a type of behaviour on its own, but rather a combination of behaviours.

3) Environment Like in multi-agent systems, agents in an agent-based model are situated in an environment. It provides the conditions under which the agents exist [63]. It defines both how agents can interact with each other and with the environment and provides information that agents can perceive. Often agents are situated in a spatial environment and can only perceive a small part of their environment. Thus, agents often have a partial view on their environment [58].

4) Agent interaction Agent-based models consist of multiple agents. A key element in these models is that the interactions between agents is modelled explicitly [58]. There are two ways agents can interact with each other: direct communication and indirect communication.

In direct communication, agents can set up a connection with another agent which acts as dedicated communication channel to exchange messages. In direct communication, agents know the receiver of the message.

In indirect communication, agents exchange information through the environment. This can be done by either triggering an event, which can influence the environment to change its state, such that the change can be perceived by other agents. Who receives the event, or perceives the change in the environment, is not known to the original sender.

3.1.2 Environment

The environment comprises the global state or the facts of the system. To study the environment's influence, we need to be able to control the environment and adjust certain parameters. Therefore we need to explicitly model the environment in an agent-based model.

Odell et all. [63] identified that the environment consists of two components: the physical environment and the communication environment.

The physical environment is defined as: *"The physical environment provides those principles and processes that govern and support a population of entities"*. The physical environment keeps track of the location of each of the entities within the environment and provides information about the physical properties of the system. It also defines physical interactions, such as collisions.

The communication environment is defined as: *"The communication environment provides those principles, processes, and structures that enable an infrastructure for agents to convey information"*. The communication environment defines how agents can exchange information amongst each other.

Russel and Norvig [74] defined several properties of an environment. Each property defines a spectrum. These properties are:

- Accessibility: Accessibility refers to the accuracy with which agents can sense an environment. The more accessible an environment is, the more accurate and up-to-date sensed data is.
- Determinism: This feature refers to the predictability of the outcome of actions. In a deterministic environment, the effect of actions is predictable. In a non-deterministic environment, multiple factors can influence the outcome of an action and the outcome is not entirely predictable. The Influence-Reaction model [56] aims to represents these uncertainties. We will discuss this model in more detail later in this section.
- Dynamism: An environment can either be static or dynamic. In a static environment, the changes in the environment only occur as a consequence of the actions of an agent. In a dynamic environment, changes in the environment can happen independent of agent actions.
- Continuity: Continuity refers to the continuity or discreteness of an environment.In a continuous environment, the state of the environment is a continuous function and agents can observe their environment continuously. In a discrete environment, the environment's state changes on discrete time steps and agents observer their environment on discrete time steps.

3.1.3 Logical time

A key feature of agent-based models that sets it apart from multi-agent systems is that time is virtualized. As discussed in Section 2.2.2, there are three different types of virtual time: discrete-time, discrete-event, and continuous time. The agent-based modelling paradigm does not specifies which representation of time should be used. Therefore, we analyzed in the current tools which representations are used.

Discrete time

This implementation of time is very popular in ABM tools. As seen in Table 3.1, all tools that were examined for this study either provide a built-in way to create discrete-time models or at least allow modellers to implement it with built-in artefacts.

We argue that the reason for the popularity of discrete time in ABM tools is twofold. Firstly, agent-based models are often very complex due to the high amount of interacting entities. The intuitivity of discrete time modelling facilitates the modelling process and allows modellers to focus on the inherently complex aspects of agent-based modelling. Secondly, tools such as NetLogo and Repast are specialized for developing models of mobile agents. The position of the agents is a continuous value through time, however, the decision making process (and internal state) adheres more to a step-wise execution. Because discrete time models are suitable for approximating continuous systems, they are a good compromise to model mobile agents.

Table 3.1: Tool support for discrete time models in ABM tools

	Only option	Default option	Implementable	Infeasable
NetLogo		✓		
Repast		✓		
TurtleKit	✓			
MASON			✓	
GAMA/GAML	✓			

Discrete Event

In Table 3.2, we see that the support for discrete event modelling in ABM tools is far less than for discrete time modelling. However, we argue that this paradigm is the most appropriate way to develop agent-based models, since it resembles the real system more accurately. In a discrete time model, agents can only perform state transition on predefined time steps. In a discrete event model, every agent can deliberately choose when it will transition its state. Also, every discrete time model can be approximated by a discrete event model, in which the time to the next state transition is the same for every component for every state. This gives the modeller more options to model the system in the most appropriate way.

Continuous time

Continuous time modelling is rarely seen in combination with agent-based modelling, if at all. We did not come across a tool that features a continuous representation

Table 3.2: Tool support for discrete event models in ABM tools

	Only option	Default option	Implementable	Infeasable
NetLogo			✓	
Repast			✓	
TurtleKit			✓	
MASON	✓			
GAMA/GAML				✓

of time and state. We argue that the reason for this is that an equation based model that takes into account the complex interactions of the agents, would be too complex to develop. Also the behaviour of an agent is not very suited for this type of modelling. At certain points in time an agent will perceive its environment, update its internal state and perform a certain action. This behaviour relates much more to the previously discussed paradigms.

Scheduling

Apart from the virtual representation of time, agent-based models face a different problem. Due to the high number of entities that need to be simulated, it is possible that multiple entities change there state simultaneously. In discrete time and discrete event models, this is done by defining an order in which the agents can change their state within a single iteration of the simulation loop. The correct choice of scheduling policy is critical in agent-based modelling, since all entities interact with the same component, the environment. If a different order of execution results in a different global state, it means the chosen policy is not the most appropriate one.

One way to define the schedule is by forcing the modeller to explicitly define the order in which the agents are executed. The classic DEVS formalism [92], for example, features the *select* function. This function is used as a tie-breaker when multiple models have scheduled a transition simultaneously. While this approach gives a lot of control to the modeller, there is a problem. If two transitions are scheduled simultaneously and transition T_1 has precedence over T_2, when T_2 is computed it can see the changes made by T_1. When the order in which models execute their transition changes the output of the simulation, the chosen scheduling policy is not the most appropriate choice. In our demonstrative example, if a hungry wolf and a hungry sheep are on the same patch, the order in which they eat matters, if the wolf has precedence, the sheep will not eat the grass. If the sheep has precedence, it eats grass first, which might result in the migration of the remaining sheep.

Agent-based tools, such as NetLogo and Repast, use a similar approach. However, they abstract the explicit ordering. Here, each time-step, all agents are shuffled randomly before execution. The reason for this choice is simplicity. The modeller does not have to be concerned about defining an order. However, the earlier discussed

problem still holds. Although it is a popular choice in current ABM tools, it is not the most appropriate one.

A third option is featured in the parallel-DEVS formalism [92]. Here, the internal state changes of the models are performed in an aribitrary order, then all external influences are gathered and are resolved after all scheduled transitions have been resolved. This results in a two phase method. In the first phase all models execute their state transition without seeing any intermediate results. In the second phase all models notify their change to the appropriate listeners.

3.1.4 Dynamic structure

A distinct property of agent-based modeling and simulation is its dynamic structure. Throughout a simulation, the population of agents can vary over time. Agents can give birth to new agents [58], which results in a growth of the population. Agents can also die, as can be seen in the predator-prey example in [78]. Through birth and death, the structure of the model can change.

3.2 Implementation of the ABM paradigm in current tools

In the previous section, we identified the different key elements in an agent-based model. In this section, we will analyze how these elements are implemented in current tools. The review is NetLogo and SARL is partially based on our previous literature study [47].

3.2.1 NetLogo

NetLogo [79] is a platform that features a graphical user interface, an integrated development environment, and a domain specific language for developing and simulating agent-based models. The tool is originally inspired by StarLisp and Logo and is widely used in research and educational fields [70].

Agents In NetLogo, agents are the central entity. Not only do they make up the population of the model, agents are also used to represent the environment as well as direct the simulation and provide information to the user. There are four types of agents in NetLogo: Turtles, Patches, Links, Observers.

Turtle agents are entities that move through the environment and represent the population of the model. They feature built-in methods to observe and move through their surroundings and modellers can define new specific breeds of turtles. Each breed has a set of variables, which make up the state of the agent.

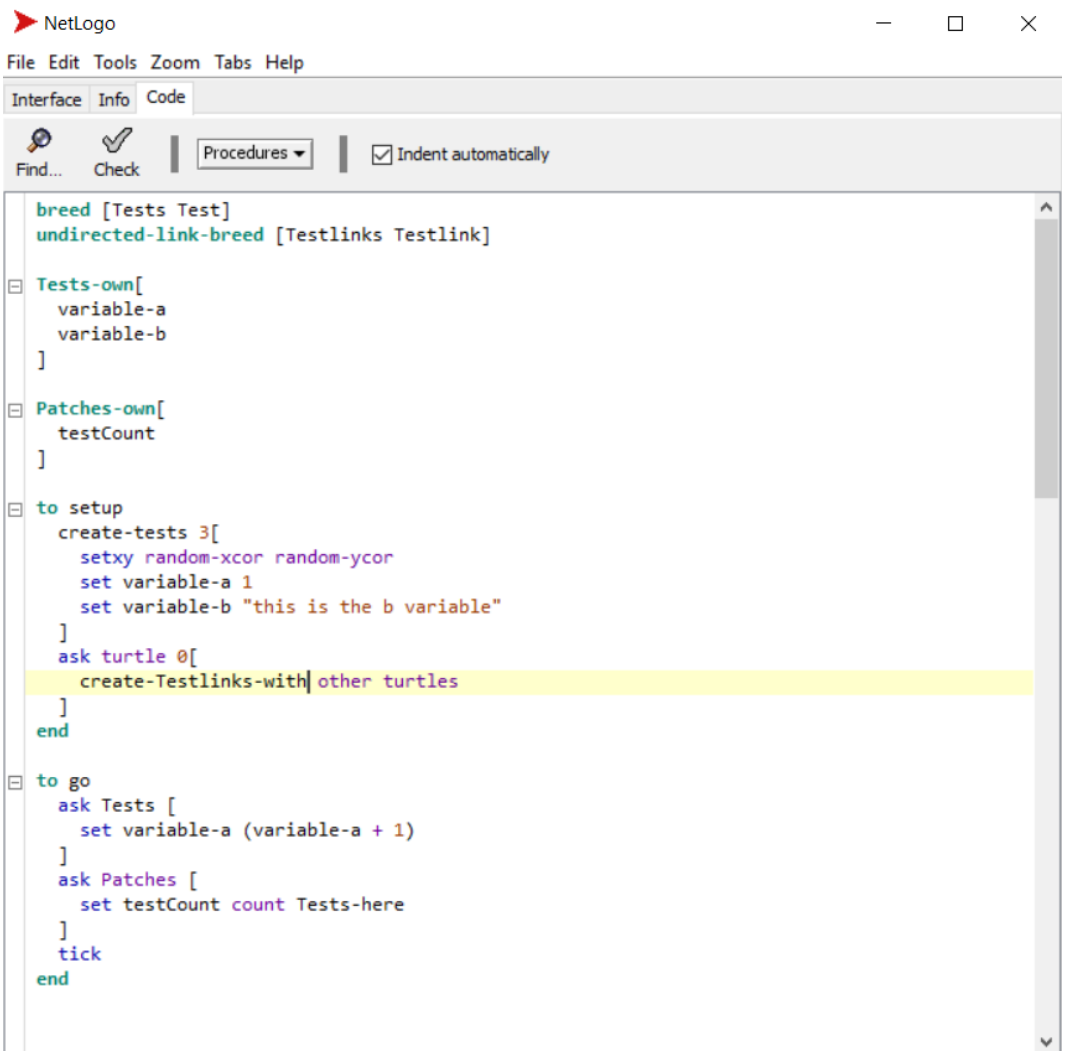


Figure 3.2: A NetLogo example containing patches, turtles, and links

Patches make up the grid in which the turtles can move. Though agents have a position with infinite precision, the environment is split up into a toroidal grid, where each of the cells is a Patch agent. Unlike turtles, users can not define custom patch breeds (for example a Road patch or a Field patch). However, we can define the fields that make up the state of the patch with the patches-own statement.

Links are used to link agents together. This allows users to create graph based environments. Links are either directed or undirected and link only two agents. Links are treated as agents.

In each model there is only a single Observer agent. The script that is written by the modeller is the implementation of the observer agent. This entity is responsible for directing all other agents by issuing instructions to other agents with the ask

statement. The ask statement has two parameters: an agent set and a block of code. Then for each agent in the agent set, the block of code is executed. The order in which the agents are chosen from the set is randomly determined. Agent behaviour is directed by another entity, which means that agents are not represented as autonomous entities. NetLogo does not have a specific way to represent the communication of agents with messages. Agent interaction is realized with nested ask statements.

Environment The environment in NetLogo is very limited. This is due to NetLogo's philosophy that everything is an agent. The environment in NetLogo is a purely spatial environment, meaning that its only task is to keep track of the agents' position and movement. The environment of NetLogo can either be seen as a continuous 2 dimensional environment, since the agent's position is an arbitrary coordinate with infinite precision, as well as a grid environment, since each agent is positioned within a patch that represents a cell in a grid.

Time implementation NetLogo appears to implement a discrete time-base, since it keeps track of a tick counter. A special command, tick, is provided, which increases the tick counter of the simulation with a single time unit or tick. Although, the most common way to implement a NetLogo model is to define a go method that defines how the overall state of the model changes at each time-step and increases the tick counter with the tick command. However, NetLogo is capable of much more. How much the time advances can be defined each simulated time-step, allowing modellers to implement a discrete event simulator within the NetLogo model. The domain specific language of NetLogo therefore resembles more a general purpose language.

Regarding the scheduling policy on a time-step level, NetLogo randomizes the agentset. When an ask statement is executed for an agentset, agents randomly take their turn and execute the body of the ask statement. Again, different scheduling policies can be implemented by the user.

Similar tools The popularity of NetLogo can be seen in the numerous tools that draw inspiration from it. A popular alternative of NetLogo for developing agent-based models is RePast. Though RePast is more of a framework that can be used as a starting base for developing agent-based models, it also features a domain specific language called ReLogo, which is based on the (Net)Logo language. As for the agents, we see the same types of agents (Turtles, Patches, Observers, and Links) with identical semantics.

The only noteworthy difference we identified between the tools is that ReLogo defines agents as Java classes and therefore makes ReLogo models more modular that NetLogo models. Though this is merely an implementation detail, modellers have more options of hiding variables and defining agent specific methods that represents agent behaviour.

Another tool that is influenced by the logo family is TurtleKit [57, 54]. TurtleKit is originally developed as a simulation platform based on the MadKit multi-agent system platform. In TurtleKit, there are patches, observers, and turtles like in the previous tools. However, the behaviour of the agents is not defined within the observer agent, rather agents implement certain methods which return a string of the next action to perform. Though the behaviour of the agents is defined within the agents itself, they have a complete view and control of the whole model state. Again we see that autonomous behaviour is not enforced or even suggested in an agent-based modelling tool.

A novel feature TurtleKit adds is agent roles. The agent-group-role organization [31] is an organizational structure for multi-agent systems that does not impose any architecture on the agents itself. A group is set of agents sharing some common characteristics. A role is a abstract representation of a function the agent has within the group. In TurtleKit, the implementation of this structure is rather limited. A role is a field of each turtle that can be used to identify a set of agents. Turtles can have multiple roles and add or drop them throughout the simulation. Groups are not present in the platform.

3.2.2 GAMA/GAML

GAMA is a platform that specializes itself in developing and simulating spatially explicit multi-agent models. It features a complete development environment with a domain specific language for implementing models, called GAML. GAMA is similar to NetLogo in that it is a complete development environment. Although some features are similar or identical between the two tools, GAMA features some concepts for agent-based modelling that are not present in NetLogo. therefore we discuss GAMA here separately.

Agents GAMA also features turtles, patches and links, but under different names: Species, Grid species, and graph species, respectively. Also a global agent is present in each model and maintains the global state of the model. Different types of agents are defined as species. A species definition consists of a set of variables, which make up the state of the agent, a set of actions the agent can perform, and a set of reflexes. An action is a composite block of statements that the agent can execute as a single action. A reflex is a block of code that is executed every time step by the agent. A reflex can also be given a condition as parameter. In this case, the reflex is only executed when the condition evaluates to true.

To influence other agents within the model, the ask statement can be used as in NetLogo. The ask statement is executed unconditionally, thus autonomy is again not enforced.

Thusfar, we elaborated on the default way to implement species in GAML. However, GAML features different control architectures to implement the behaviour of agents.

```
/***
* Name: NewModel
* Author: Leys
* Description:
* Tags: Tag1, Tag2, TagN
***/

model NewModel

/* Insert your model definition here */

species TestSpecies {
    float var_a <- 0.0;

    bool increase_a <- true;

    action stop_increase{
        increase_a <- false;
    }

    reflex increase_a when: var_a < 0{
        if(increase_a){
            var_a <- var_a + 1;
        }
        if(var_a > 100){
            do stop_increase;
        }
    }
}
```

Figure 3.3: An example of a GAMA species

These control structures are: finite-state machine, task-based, rule-based, and user control. When a species has a finite-state machine control architecture, its behaviour is defined as a finite-state machine. The modeller can define a set of states and state transitions that are triggered when a condition is met. In a task based architecture, the behaviour is defined as a set of task, each time step one or several tasks are executed according to their weights. In a rule-based architecture a set of rules are defined. Each time step, all rules are evaluated, if the left-hand side is evaluated to true, the right-hand side of the rule is evaluated. In the user-control architecture, a user interface can be defined to manually control the agents.

Environment As mentioned in the previous section, GAMA features the grid species, as well as the graph species to represent both spatial environments as well as link-based environments.

GAMA has a lot of built-in plug-ins or assets that can be used to create very extensive environments. An example is the built-in species, the physical_world species. This is a base for implementing entities that act within a 3 dimensional world with real life physics.

Though the support for physical environments is extensive, the communication environment is lacking. All communication between agents should be done with the ask statement.

Time-Base GAMA uses a discrete-time implementation. In contrast to NetLogo, the size of ticks can not be altered. Though some of the different control architecture might give the impression that time advances contiuously or in a discrete event-matter, behind the scenes the state of the model is always updated on equidistant time-intervals.

The scheduling policy during one time-step can be chosen for each species separately. Also, it can be defined globally in which order the species are executed.

3.2.3 MASON

MASON [49] is a framework for building models with large amount of individual entities. As opposed to RePast, MASON requires more experience with general purpose programming. MASON also does not feature agents as a first-class abstraction, which indicates that the purpose of MASON is to provide a basis for developers to create all sorts of models where large amounts of individual entities need to be simulated.

Agents As mentioned, MASON does not feature agents as a first-class abstraction. To represent entities, MASON provides the steppable interface. Each class that implements this interface can make use of MASON's scheduler. The Steppable interface only has a single method called step. Whenever the scheduler executes a scheduled entity, it will call its step method. The step method also receives the full model state as parameter, which means that during the step, an agent has control over the complete state of the model.

The minimalistic approach of implementing agents entails that agent properties are not enforced by the framework. Autonomy is not guaranteed, since agents are simple java classes. There is no built-in way of sending messages between agents and no action model is implemented which indicates how the agents can act upon their environment.

Environment In MASON a spatial environment is called a field. MASON also provides a library of predefined field implementation. Examples of these are Continuous 2D environment, a sparse grid, a graph based environment, ... For each of these environments there is functionality to visualize them, as well as the entities within the environment. On top of this, inspectors can be used to inspect properties of the environment.

Time-Base MASON implements a discrete event time-base, where all steppable entities can schedule events at arbitrary timesteps. The scheduler is part of the model state, which also contains all agents, the field or environment, and a random number generator. State transitions (or Events as they are called by the documentation) can

be scheduled either once or repeating, however if an agent dies at a certain time and is still scheduled to perform an event, an error occurs. This means that the correct implementation of dynamic populations is up to the developer.

If multiple entities scheduled a transition at the same time-step, the entity that scheduled its transition first (wallclock time) has precedence and will execute its transition first.

Similar tools Though MASON has an extensive list of features, a lot of implementation details are left to the modeller. MASON therefore can be seen as a framework or library that acts as a foundation for developing and simulating agent-based models, rather than a tool that presents formalism that encapsulates all facets of agent-based modelling.

MASON is not the only option for such a library. The classic version of RePast [23] is very similar. RePast [23] started its lifetime as a java implementation of the SWARM framework, but ultimately decided to go its own way [70]. SWARM was originally a framework for developing and simulating models with large amounts of individual entities (such as agent-based models). RePast features most of SWARM's functionality, but adds more options for collecting and visualizing data from the models. One of RePast flag-ship features is its ability to run simulations in batch. RePast claims to be discrete-time by representing time as ticks, however, more complex time-bases are implementable within the framework itself. Furthermore, the workflow is rather similar to that of MASON.

3.2.4 SARL

SARL is an agent-oriented programming language. This means that it is a general purpose programming language that uses concepts from multi-agent systems as first class abstractions. SARL is a platform for developing multi-agent systems and thus has no modelling and simulation functionality. However, due to its agent based syntax and semantics, we were interested if a tool like SARL can be used as a modelling and simulation tool.

Agents SARL features agents as first class abstractions. Similar to class definitions, agent types can be defined by agent definitions. Similarly to class definitions, developers can define a set of data fields, which make up the state of the agent. Part of this state can be used to implement the local representation of the agent's environment or its beliefs. Also agent methods can be defined. In SARL a class method can be interpreted as an action or an atomic event that the agent can perform. Agent methods can only be private and thus, no external entity can directly control the agent. This assures that agents are autonomous. To collaborate, agents can send events to each other. An event is an object that can be broadcasted to all

reachable agents or to a single agent. To react to events, SARL has the *on* statement. In the on statement, developers can specify how the agent reacts to certain events. Additionnally, a guard can be provided, such that agent only reacts to the event if the guard evaluates to true.

Environment Unlike all previous mentioned tools, SARL provides a communication environment in the form of spaces. Spaces define how the agents within the space can interact with each other. Although the implementation of a space is up to the developer, SARL provides the EventSpace as a built-in space. In the EventSpace, events can be emitted to all participants or to a specific scope.

SARL does not feature any built-in way to represent a physical environment. This is due to its purpose. When developing a multi-agent system, the physical environment is made up of real world physical elements that influence the agents. Only when developing agent-based models, an explicit virtual representation is necessary.

Time-Base Since SARL does not feature any modelling and simulation functionality, it also does not have a virtual representation of time. SARL programs are deployed in a real time application and can only access system time.

3.2.5 Discussion

After inspecting a series of tools, we concluded that autonomy is never enforced in agent-based modelling tools. This is odd, given that in the literature, autonomy is regarded as the a aspect of agency [51, 43, 58]. None of the dedicated tools even define a standard way of how agents can interact with each other or their environment other than directly invoking messages. We argue that this is phenomenon is a deliberate design choice to make the implementation of the models more easy. When we invoke methods directly, we are certain of the changes in the state of the model after the execution of the method. This is much easier to reason about then when an agent does not know the outcome of his actions. The complexity of agent-based models is already very high due to the large amount of heterogeneity [14].

Not enforcing autonomy, however, is a divergence from the paradigm. By looking at SARL, we got an idea of how autonomy can be assured. SARL does this in a rather elegant way, by allowing only private methods and providing a dedicated way to react to external input. Workarounds that violate the autonomy property can be implemented, however, this is far from trivial. We therefore consider SARL's implementation of agents very suited to represent them.

Because of this we will use SARL as a starting point to implement our platform with precise semantics. Another reason is SARLs versitality as general purpose language.

3.3 Existing formal semantics

To develop our own formal semantics, we analyzed the state of the art regarding existing attempts to formalize agent-based modeling and simulation . In this section, we will discuss the most prominent attempts to formalize the semantics of ABM and identify the parts we wish to reuse in our formal semantics.

3.3.1 Partial formal semantics

Not all formal semantics proposed for agent-based modeling and simulation cover the complete paradigm, but rather focus on a specific concept. In this section we discuss some of these that seemed most useful for us.

The IRM4S

The Influence Reaction Model for Simulation or IRM4S [56] is a model developed to express how agents can interact with their environment. In previous interaction models, the modifications of a model on its environment are direct (an example can be found in [35]). However, such an action model presents certain complications. Firstly, actions need to be handled individually, which complicates the implementation of simultaneous events. Secondly, modellers are unable to model uncertainty in the external dynamics of the agents. An agent should not be able to predict the result of his actions, because it does not have an exhaustive knowledge about the environment's settings. Especially when multiple actions happen simultaneously (e.g. if two agents push on both sides of door, the agents can not predict whether the door will open or not). Thirdly, the autonomy of environmental entities is comprised. Whenever an agent performs an action on the environment, it resolved without respecting the goals of all entities involved, even if these entities are autonomous. Finally, the implementation of the model specifications should not influence the results [92]. However, the classical action model is very sensitive to implementation details [55].

The Influence Reaction Model aims to solve these problems by replacing direct actions by influences and reaction. Instead of performing an action, an agent produces influences, which define the intentions of an agent (e.g. open a door). When the environment knows all influences that are produced during a time-step, it can compute its reactions. In the IRM4S, we still have the world state Σ and, in addition, a dynamic state $\delta \in \Delta$. δ is represented by a tuple $\delta = \langle \sigma, \gamma \rangle$, where $\sigma \in \Sigma$ is the set of environment variables and $\gamma \in \Gamma$ is the set of influences. And the system's evulotion is defined as a two phase mechanism:

$$\textbf{(1)}\ \gamma'(t) = Influences(\sigma(t), \gamma(t))$$

$$\textbf{(2)}\ \delta(t+dt) = \langle \sigma(t+dt), \gamma(t+dt) \rangle = Reaction(\sigma(t), \gamma'(t))$$

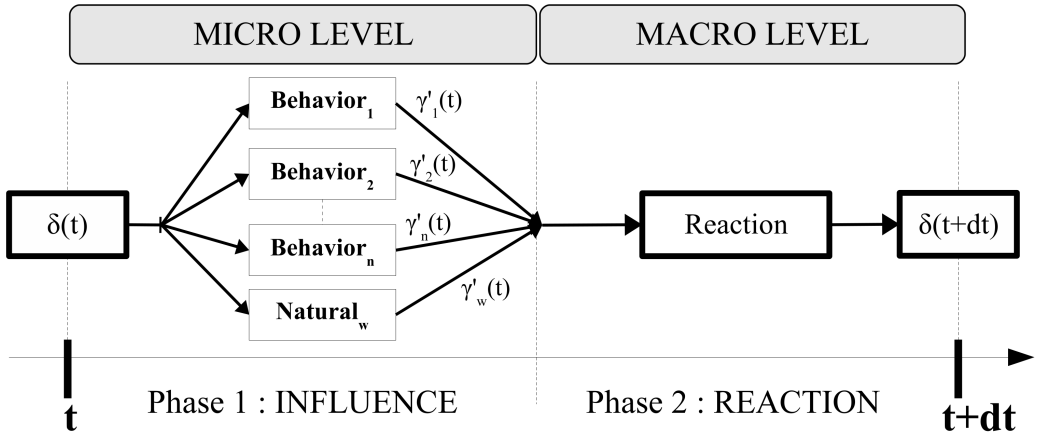


Figure 3.4: System evolution in the IRM4S model [56]

For the first phase, an agents behaviour is defined as a function $Behaviour_a : \Sigma \times \Gamma \rightarrow \Gamma'$ which can be decomposed into:

$$p_a(t) = Perception_a(\sigma(t), \gamma(t))$$

$$s_a(t + dt) = Memorization_a(p_a(t), s_a(t))$$

$$y'_a(t) = Decision_a(p_a(t), s_a(t))$$

Where $y'_a(t)$ are the influences produced by agent a at time-step t. Since a dynamic environment should be able change its state without agent influences. Therefore, the IRM4S model features the $Nature_w : \Sigma \times \Gamma \rightarrow \Gamma'$, similar to the $Behaviour_a$ function, that allows modellers to implement the environment's dynamics. This function produces the influences of the environment on itself, such that we have:

$$y'_w(t) = Nature_w(\sigma(t), \gamma(t))$$

The total set of influences that are present at time-step t, $\gamma'(t)$ are defined as:

$$\gamma'(t) = Influence(\sigma(t), \gamma(t)) = \{\gamma(t) \cup \gamma'_w(t) \bigcup_a \gamma'_a(t)\}$$

The second phase is the reaction phase. The semantics of the Reaction is that it computes the subsequent state of the environment, based on the current state and the influences. However, due to the high heterogeneity of influences, IRM4S does not propose a unique solution. The implementation of the system's dynamics is up to the modeller.

Belief-Desire-Intention

The belief-desire-intention model [36] is a model that tries to formalize cognitive behaviour of agents. This model originally comes from the field of AI and is used to develop multi-agent systems. However, due to the increase in complexity in agent-based models, it can be an elegant way of modelling the behaviour of agents. As seen in [3], agent-based modeling and simulation tools have been developed that enable the modelling of BDI-agents.

The beliefs of an agent is the knowledge the agent has on its environment, other agents, and itself. Note that this is only local information. If an agent only perceives a part of its environment some of the beliefs may become outdated.

The desires of an agent are the overall goals the agent wants to achieve. The goal represents a desired end state of the system. This goal oriented approach allows agents to seek out alternative solutions, whenever there current solution fails unexpectedly. Altough this is more useful in multi-agent systems rather than agent-based modelling, it allows modellers to represent the real world in better detail. For example, if we want to model the behaviour of people in a building during an emergency, each person has the goal of leaving the building as soon as possible. People will deliberate different solutions to achieve their goal, rather than execute a fixed sequence of actions.

Finally, intentions can be seen as plans the agent can execute to achieve its goals. These intentions are more complex than simple actions, since they can specify a procedure that spans a longer period of time.

The belief-desire-intention model remains a very high-level model. It specifies general concepts in designing an agent's behaviour, but makes no assumptions of how it should be implemented.

3.3.2 DEVS as a foundation

The relation between DEVS [85] and ABM has been studied multiple times [8, 13, 82] and DEVS has been used as a foundation to develop a formal semantics for ABM [8, 60]. The benefits of using DEVS are twofold. First, the semantics of the DEVS formalism are formal and clear. Secondly, DEVS is a universal modelling formalism that can be to implement a wide variety of models [93]. Moreover, certain concepts of the DEVS formalism can be mapped directly to concepts in agent-based modeling and simulation . For example, the internal transition function resembles the pro-active behaviour of an agent, while the external transition function resembles reactive behaviour [60].

The Classic DEVS formalism

Classic DEVS [93, 85] features two types of models, atomic DEVS models and coupled DEVS models. Atomic models are *the indivisible building blocks of a model* [85] and can be connected in a coupled model.

An atomic DEVS model is a tuple $\langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$, where:

- X is the set of input values.
- Y is the set of output values.
- S is the set of states.
- $\delta_{int} : S \to S$ is the internal transition function.
- $\delta_{ext} : (Q \times X) \to S$ is the external transition where $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$, and e is the elapsed time since the last state transition.
- $\lambda : S \to Y$ is the output function.
- ta is a function $ta : S \to \mathbb{R}^{+}_{0,+\infty}$, which maps the internal state of the model to the time to the next internal transition.

The interpretation of an atomic model A is as follows. At any time A is in a state $s \in S$. If no external event occurs, A will remain in state s for time $ta(s)$. Note that $ta(s)$ can take on the values of 0 and $+$. Whenever $ta(s) = 0$, no external events can happen between the state transitions and we say s is a *transitory* state . When $ta(s) = +\infty$, the model will wait indefinitely until an external event occurs. Such a state is also refered to as a *passive* state [92]. When $ta(s)$ expires, A outputs the value $\lambda(s)$ and, subsequently, changes its state to $s' = \delta_{int}(s)$. The next transition is then scheduled at $ta(s')$.

If an external event $x \in X$ occurs before the expiration of $ta(s)$, A changes its state to $s' = \delta_{ext}(s, e, x)$, where e is the elapsed time since the last state transition. Similar to the internal transition, the next state transition is scheduled at $ta(s')$.

Altough we already defined how atomic models can be interrupted by external events, we have not defined who can interrupt a model. To define this, we introduce coupled models, which combine different atomic models and allow communication between them.

A coupled DEVS model consists of a set of atomic or coupled DEVS. Within the coupled model, models can be coupled by connecting their ports. Above that a coupled DEVS model can also be coupled by another coupled model. This allows the creation of hierarchical models. To facilitate the modelling process, we will first redefine the input and output values of atomic models, such that they feature ports. The changes to the previous definition are:

- $X = \{(p, v) | p \in InPorts, v \in X_p\}$ is the set of input ports and values.
- $Y = \{(p, v) | p \in OutPorts, v \in X_p\}$ is the set of output ports and values.

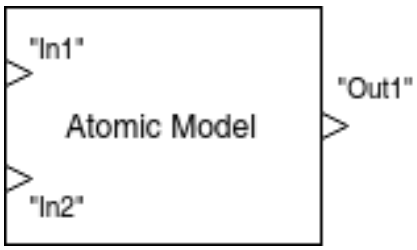


Figure 3.5: An atomic DEVS model with ports

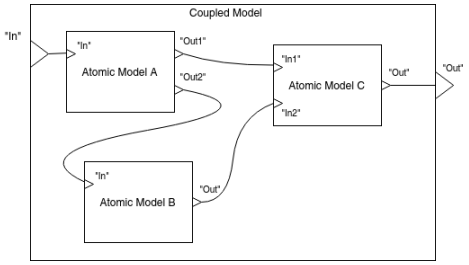


Figure 3.6: A coupled DEVS model

Then a coupled DEVS model is defined as follows:

$$C = \langle X, Y, D, M, EIC, EOC, IC, Select \rangle$$

Where:

- $X = \{(p, v) | p \in InPorts, v \in X_p\}$ is the set of input ports and values.
- $Y = \{(p, v) | p \in OutPorts, v \in X_p\}$ is the set of ouput ports and values.
- D is the set component identifiers.
- $M = \{M_i | i \in D\}$ is the set of component specifications such that $\forall m \in M$: m is a DEVS model.
- $EIC \subseteq \{((C, ip_C), (d, ip_d)) | ip_C \in InPorts_C, d \in D, ip_d \in InPorts_d\}$ is the external input coupling.
- $EOC \subseteq \{((d, op_d), (C, op_C)) | op_C \in OutPorts_C, d \in D, op_d \in OutPorts_d\}$ is the external output coupling.
- $IC \subseteq \{((a, op_a), (b, ip_b)) | a, b \in D, a \neq b, op_a \in OutPorts_a, ip_b \in InPorts_b\}$ is the internal coupling.
- $Select : 2^D \rightarrow D$ is the tie-break function.

The coupled model describes how DEVS models are interconnected. Note that we make no assumption whether the models in M is an atomic model or a coupled model. This allows modellers to create a hierarchy of models.

The Select function is used whenever multiple component models have scheduled a state transition at the same time. In that case the select function is used to determine which of these models executes its transition first.

The DS-DEVS formalism

An obvious limitation of the DEVS formalism regarding agent-based modeling and simulation is the lack of support for dynamic structures. An extension on the classic

DEVS formalism, that deals with this problem, is dynamic structure DEVS [9]. The DS-DEVS formalism comprises two types of models: basic models and network models.

The basic models are defined in the same way as atomic models in the classic DEVS formalism. While the network models are defined as a tuple:

$$\langle \chi, M_\chi \rangle$$

Here, χ is called the network executive and M_χ is the model of the executive. M_χ is a DS-DEVS basic model and is defined as:

$$M_\chi = \langle X_\chi, Y_\chi, S_\chi, \delta_{int\chi}, \delta_{ext\chi}, \lambda_\chi, ta_\chi \rangle$$

The information of the structure of the model is located in the state of the network model. A state $s_\chi \in S_\chi$ is defined as a tuple:

$$s_\chi = (X_\Delta, Y_\Delta, D, M, \{EIC, EOC, IC, SIC, SOC\}, Select, \Theta)$$

Where:

- $X_\Delta = \{(p, v) | p \in InPorts_\Delta \in X_{\Delta p}\}$ is the set of input ports and values of the DSDEVS network. Δ is the DSDEVS network.
- $Y_\Delta = \{(p, v) | p \in InPorts_\Delta \in X_{\Delta p}\}$ is the set of input ports and values of the DSDEVS network.
- D is the set component identifiers.
- $M = \{M_i | i \in D\}$ is the set of component specifications such that $\forall m \in M : m$ is a DEVS model.
- $EIC \subseteq \{((\Delta, ip_\Delta), (d, ip_d)) | ip_\Delta \in InPorts_\Delta, d \in D, ip_d \in InPorts_d\}$ is the external input coupling.
- $EOC \subseteq \{((d, ip_d), (\Delta, ip_\Delta)) | ip_\Delta \in OutPorts_\Delta, d \in D, ip_d \in OutPorts_d\}$ is the external output coupling.
- $IC \subseteq \{((a, op_a), (b, ip_b)) | a, b \in D, a \neq b, op_a \in OutPorts_a, ip_b \in InPorts_b\}$ is the internal coupling.
- $SIC \subseteq \{((d, ip_d), (\chi, ip_\chi)) | ip_\chi \in InPorts_\chi, d \in (D \cup \Delta), ip_d \in OutPorts_d\}$ is the structural input coupling.
- $SOC \subseteq \{((\chi, ip_\chi), (d, ip_d)) | ip_\chi \in OutPorts_\chi, d \in (D \cup \Delta), ip_d \in InPorts_d\}$ is the structural input coupling.
- $Select : 2^{(D \cup \{\chi\})} \rightarrow (D \cup \{\chi\})$ is the select function.
- Θ contains other state variables not defined before.

By moving the structural information to the state of a network work model, we can specify dynamic changes in the structure of the model by using the $\delta_{int\chi}$ and $\delta_{ext\chi}$. Note that there are no assumptions on which structural changes can be performed.

The Parallel-DEVS formalism

Agent-based models often consist of a large set of agents. Due to the large number of entities that are modelled, the chances of multiple state transitions occurring at the same time can become high enough that it becomes a concern of the modeller.

In classic DEVS, the order in which the agents perform their state transition is explicit and given by the *Select* function. A downside of this approach is, for example, that even though two state transitions happen simultaneously, the result of one state transition influences the other state transition, because it was executed first. An alternative DEVS-related formalism that tries to solve this problem is parallel DEVS. In parallel DEVS all external influences of a model that occur during a single time step are gathered and passed on as a bag. Formally, an atomic model is defined as:

$$\langle X_M^+, Y_M^+, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$$

Where:

- X_M is defined as in Classic DEVS models with ports and X_M^+ is the set of all possible bags over X_M.
- Y_M is defined as in Classic DEVS models with ports and Y_M^+ is the set of all possible bags over Y_M.
- S is the set of states.
- $\delta_{ext} : Q \times X_M^+ \to S$ is the external transition function. Q is defines as in classic DEVS.
- $\delta_{int} : S \to S$ is the internal transition function.
- $\delta_{con} : Q \times X_M^+ \to S$ is the confluence transition function.
- $\lambda : S \to Y_M^+$ is the output function.
- $ta : S \to R_{0,+\infty}^+$ is the time-advance function.

Apart from having a bag of inputs, parallel DEVS models have another difference, the confluence transition function. This function is called whenever a model has collision of an internal and external transition scheduled at the same time-step.

The coupled model of parallel DEVS is defined in the same way as classic DEVS, except for the Select function, which is omitted for parallel DEVS.

Müller [60] shows the relation between parallel DEVS and IRM4S. Certain concepts of the IRM4S model can be mapped directly to the parellel DEVS formalism. The influence phase at time t in the IRM4S model is equivalent to the internal transition and output step of each of the models that scheduled a state transition at time t. If M is the set of all these models. The influence at time t is given by:

$$\gamma'(t) \simeq \bigcup_m \lambda_m(\delta_{int,m}(s_m)) \text{ for } m \in M$$

We then define the set of all atomic models that receive an input value at timestep t and perform an external transition as I, and $\gamma'(t)_i$ with $i \in I$ as the bag of input values received by model i. The reaction of the model is computed as follows:

$$Reaction(t) \equiv \bigcup_i \delta_{ext}(s_i, e_i, \gamma'(t)_i) \text{ for } i \in I \text{ and } e_i \text{ is the elapsed time of } i$$

Since parallel DEVS was originally not developed to implement IRM4S or agent-based models in general, there are some parts missing. For example, there is no perception step present in parallel DEVS and the models can only "perceive" their environment by receiving input messages on previous time-steps.

The M-DEVS formalism

Until now, we only reviewed generic DEVS formalisms. Though they can be used to specify an agent based model, they were not originally designed to do so. The M-DEVS formalism [60] is the first DEVS-related formalism that was specifically designed for agent-based modeling and simulation .

In this formalism, J. P. Müller aims to address the short comings of the earlier mentioned formalisms regarding agent-based modelling and simulation. Firstly, Müller argues that the parallel-DEVS formalism is particularly suited as a starting point, because its relation with IRM4S (as mentioned above).

In the current definition of DEVS and parallel-DEVS, multiple state transition can occur simultaneously when $ta(s) = 0$. However, a physical system never performs these instantaneous state transition. This leaves the question what the meaning is of $ta(s) = 0$. Müller suggest that state transitions with $ta(s) = 0$ only occur for the diffusion of information and observation (e.g. agents perceiving the environment). The solution that is proposed is to seperate these transitions from the physical state transitions. This leads to the definition of an M-DEVS^{v1} entity:

$$\langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda_{ext}, ta, \delta_{log}, \lambda_{log} \rangle$$

Whith the same definitions as parallel-DEVS plus:

1. $\delta_{log} : Q \to S$ the logical transition function
2. $\lambda_{log} : Q \to Y^b$ the logical output function, and
3. $ta : S \to R_0^+$

Herein, δ_{ext} and λ_{ext} are used for physical state transitions, and δ_{log} and λ_{log} are used for logical influences, such as information diffusion. λ_{log} is called after each transition(even λ_{log}, to propagate the information of the new state. δ_{log} is used to react to the propagated information sent by an external entity with the λ_{log} function.

An inherent property of an agent-based model is the dynamic structure. DS-DEVS defines how these structural changes can be defined in a DEVS context, however the

structural changes of agent-based models are more specific. A property mentioned by Michel [58] states that only agents can be responsible for the creation of other agents. As it is also seen in several ABM and MAS tools [72, 79, 23], agents are also responsible for their own death. To deal with this, agents need to be able to send specific influences which should be separated from external influences and internal influences. The complete definition of an M-DEVS entity is:

$$\langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \delta_{log}, \lambda_{ext}, \lambda_{int}, \lambda_{log}, \lambda_{str} \rangle$$

With the same definitions as for an M-DEVS^{v1} entity plus:

- $\lambda_{int} : S \to (\mathbb{R}^{+}_{+\infty} \setminus 0) \times I$ where I is the set of internal influences.
- $\lambda_{str} : S \to Y^{+}$ is the structural output function.

As for the structure of a coupled M-DEVS formalism, M-DEVS adopts the Dyn-DEVS approach by A. Uhrmacher et al. [81]. In this approach, each entity is allowed to specify the topology in a formalism called DynDEVS. However the Dyn-DEVS approach does not respect the closure under coupling property that is shared amongst all DEVS formalism. This property allows coupled models to be expressed as atomic models, eliminating the need for defining their formal semantics separately. This makes the M-DEVS formalism not a real DEVS extension, but rather a DEVS inspired formalism.

The LDEF formalism

Another ABM oriented formalism that finds its roots in the DEVS formalism is LDEF (Large-scale, Dynamic, Extendible Formalism) [8]. The developers of LDEF first analysed the relation between ABM and several DEVS formalisms (Classic DEVS, Mobile DEVS, Cell-DEVS), by defining the mapping between the general terminology in agent-based models and concepts in DEVS-oriented formalisms, Table 3.3.

Alongside DEVS-oriented formalisms, a set of formalisms from the AI field have been studied, such as the environment model of Ricci et al. [71]. After analization, Bae et al. proposed their own formalism for ABM, called LDEF.

First, we describe the elements of the LDEF formalism. There are two types of elements in the LDEF formalism to describe the behaviour of entities and the structural dynamics respectively. The behavioural elements are, in turn, categorized in two types: Action models, who model agent behaviour, and environment element models, that model the behaviour of other elements in the environment. The action model is defined as a tuple:

$$\langle X, Y, S_{aw}, S_{cond}, S_{act}, P, D, A, ta \rangle$$

Where:

Table 3.3: Mapping the general terminology in the ABM contexts to elements in DEVS-oriented formalisms [8]

Categories of tuples in formalisms		DEVS oriented formalisms		
		DEVS Formalism	Mobile-DEVS formalism	Cell-DEVS formalism
Agent	Input or ***Perception***	Input event set	Input event set, Structure change events set	Input event set, Input event values
	Output or ***Action***	Output event set	Output event set	Output event set
	State	State set	State set	State set and values, Delay, Neighborhood size
	State transition or ***Decision***	External transition function, Internal transition function	External transition function, Internal transition function	External transition function, Internal transition function, Local computation function
	Output function or ***Action***	Output function, Time advance function	Output function, Time advance function	Output function, Transport delay, State's duration function
Environment				Cell space, Border cells
Multi-Agent Coupling Structure or ***Interactions***	Component or ***Population***	Component model set	Activated model set, Model activation function	Neighborhood set
	Coupling Relations or ***Neighborhood, Boundary***	External input couping relation, External output coupling relation, Internal coupling relation	Structure state set, Structure transition function, External input coupling relation, External output coupling relation, Internal coupling relation, Change coupling relation	Input coupling list, Output coupling list, Translation function

- X is the set of input events
- Y is the set of output events
- S_{aw} is a set of situation awareness state, which contain information about the external information of the agent.
- S_{cond} is a set of condition states, which describe conditions for the agent's decision-making.
- S_{act} is a set of action states, which indicate which action the agent will perform.
- $P : X \times Q$ is the perception function, where $Q = \{(s, e)|s \in S_{aw}, 0 \leq e \leq ta(S_{act})\}$
- $D : S_{aw} \times S_{cond} \times S_{act} \to S_{act} \times S_{cond}$ is the decision function.
- $A : S_{act} \to Y$ is the action function
- $ta : S_{act} \to \mathbb{R}^+$ is the time advance function.

Herein, the perception and decision function replace the state transition functions in an atomic DEVS model, while A is used as the output function.

As for the environment element models, LDEF adopts the definition of atomic DEVS models. Thus an environment element model is defined as:

$$\langle X, Y, S, q_{init}, ta, \delta_{int}, \delta_{ext}, \lambda \rangle$$

Where:

- S is a set of internal states.
- X is the set of input events.
- Y is the set of output events.
- $q_{init} \in S$, the starting state.
- ta is a function $ta : S \to \mathbb{R}^{+}_{+\infty}$, which maps the internal state of the model to the time to the next internal transition.
- δ_{int} or the internal transition function is a function $\delta_{int} : S \to S$ that maps each internal state to the subsequent internal state.
- δ_{ext} or the external transition is a function $\delta_{ext} : (Q \times X) \to S$ where $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$, which given an input message retrieved in state s before its ta(s) elapsed returns the next state.
- λ or the output function is a function $\lambda : S \to (Y \cup \phi)$ which maps each internal state to an output message which is sent via a certain output port. This message can be the empty message *phi*.

To couple these elements into an agent-based model, LDEF provides structural elements. These elements are called dynamic structural models (DSMs) and are defined as:

$$\langle X, Y, M, S_{CS}, sCS, \delta_{CS} \rangle$$

Where:

- X is a set of input events.
- Y is a set of ouput events.
- M is a set of component models.
- S_{CS} is a set of coupling states. In our understanding, a coupling state is used as an identifier for a certain topology.
- $sCS \subseteq \mathrm{S}_{CS} \times CS$ where $CS = \{M_s, EIC_s, EOC_s, IC_s, SELECT_s\}$ defines a set of structures or topologies of the model. The components of CS are defined similarly as in the DEVS formalisms.
- $\delta_{CS} : \cup_{m \in M_s} m.Y \times S_{CS} \to S_{CS}$ is the coupling structure state transition. Based on all outputs of the models in the current topology, a new topology is determined. $m.Y$ is the output events of m.

The DSMs are LDEF's alternative to the coupled model in the DEVS formalism and is used to couple a set of behavioural elements. In turn these DSMs can be coupled by other DSMs. These structural elements are used to represent certain concepts in agent-based models, as seen in Figure 3.7. The root model is called the agent-based model and contains both the agent models as well as the environmental models. The leaves comprise all behavioural elements, and their parents consist of

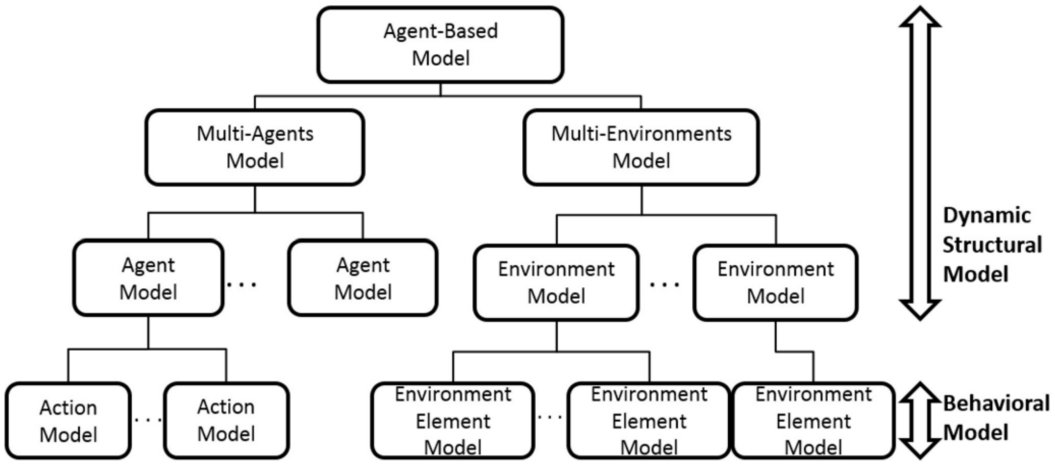


Figure 3.7: The hierarchical structure of an agent-based model in LDEF

agents and environmental models. Agents and environmental models can then be grouped into Multi-Agent models and Multi-Environment models.

3.3.3 Discussion

After analyzing the different formalizations related to agent-based modelling and simulation, we will now discuss our opinion of said formalizations. We will identify both strong points as well as weaker points.

IRM4S

We consider the IRM4S model [56] an excellent model to represent how entities in an agent-based model can interact with each other. Firstly, it provides an elegant solution to the parallelism inherent to agent-based models. These models often consist out of a large set of agents and simultaneous execution of actions is an important concern. Secondly, the IRM4S resembles real world interactions more convincing than traditional action models. For example, if two agents want to push simultaneously on a door from opposite sides, traditional action models will give one of the two agents precedence over the other and the door will open to one side. The IRM4S considers the influences of both agents and will keep the door closed, as the resulting force of both pushes is zero.

Belief-Desire-Intention

The Belief-Desire-Intention model [36] is good model for modelling complex behaviour. The goal directed behaviour that BDI advocates allows modellers to realistically model the behaviour of entities, which is a very desired property in agent-

based modeling and simulation . However, the model keeps a very high level point of view and it can be realized in a variety of approaches. For the formal semantics that we wish to implement, we will show how a BDI behaviour can be implemented in the formalism.

Classic DEVS, DS-DEVS, and Parallel-DEVS

The classic DEVS formalism several advantages regarding the implementation of agent-based modelling. Firstly, the semantics of classic DEVS have been formally defined [92]. Secondly, Atomic models are autonomous in that they can not be controlled directly by external entities. Finally, the internal transition function and external transition function have similar semantics to proactive and reactive agent behaviour respectively. This makes DEVS a good foundation. However, some parts are still missing.

The first concern is dynamic structure. In the predator prey model, both populations can grow and schrink due to the birth of new animals and the death of existing animals. In classic DEVS, however, we can not create or delete models during the simulation. The DS-DEVS formalism provides a solution for this problem.

M-DEVS

Müller stresses the relation between concepts in the classic DEVS formalism, as well as suitedness of P-DEVS to implement IRM4S. Both are very valuable contributions, since they support our claim that discrete event time-bases are most suited for agent-based modelling. However, in the defintion of M-DEVS certain design choices where made that seem rather odd.

M-DEVS make a distinction between logical and physical transition to prohibit instantaneous state transitions (transitions with ta(s) = 0). Their argument for this is that in a real world physical systems, instantaneous state transitions do not occur and, in models, instantaneous state transitions are only used for the distribution of data. We, however, argue that there are other reasons for allowing instantaneous state transition. The most obvious reason is that the time it takes to perform the transition is so small, that it is insignificant to the properties that are studied with the model. In this case, it is perfectly fine to allow instantaneous state transitions.

M-DEVS also introduces an internal output function, that replaces the time-advance function of classic DEVS. Instead of performing an internal state transition, agents react to internal events. This change is purely cosmetic and does not make the formalism more clear. We argue that it only confuses modellers who are familiar with the DEVS formalism.

LDEF

The LDEF formalism [8] appears to be a very in depth formalism that aims to capture the essence of agent-based modelling and simulation. The process of an agent is divided into perception, decision-making, and acting. As opposed to M-DEVS, LDEF considers the difference between agents and environmental entities. LDEF also formally defines how structural dynamics can be implemented. Though LDEF is very thorough, there are some limitations. LDEF does not allow models to output a bag of influences, but adheres to the classical action models that treat actions separately. Another remark is that agents are not a first class abstraction. Agents are defined by a recurring construct. This seems rather odd, given that agents are the central concept of an agent-based model. The same argument holds for the environment.

3.4 The SARL meta-model

In the previous chapter, we showed how current ABMS tools implement the paradigm. Of these tools, SARL [72] has the best representation of an agent. In this section, we will go into more detail about the SARL language and discuss its meta-model, which can be seen in Figure 3.8.

SARL is a general purpose programming language. Apart from agent specific concepts, it also features a basic support for object oriented programming, such as classes, interfaces, inheritance, etc. On top of that, SARL and Java are 100% interoperable. Since this part of SARL is less relevant for an agent-based modeling and simulation platform, we will focus on the multi-agent part of SARL.

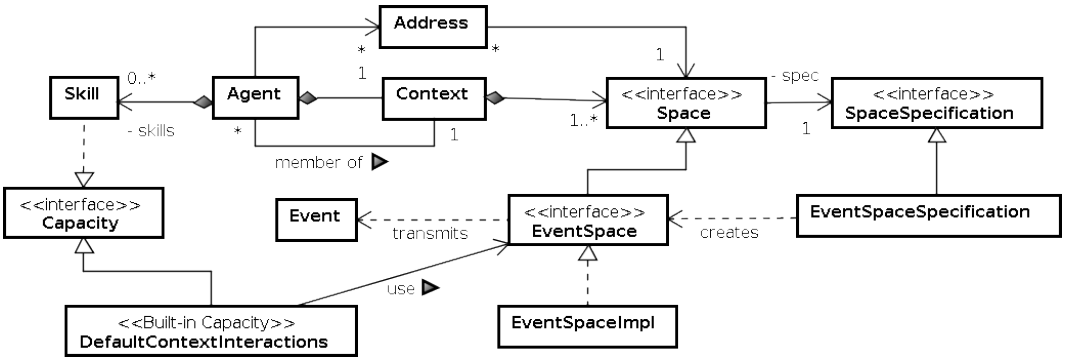


Figure 3.8: The SARL meta-model

Contexts and Spaces

SARL does not have a concept of environment. Environments are replaced by Contexts and Spaces.

A Context is defined as: *A Context defines the perimeter/boundary of a subsystem, and gathers a collection of Spaces [72]* All agents within a context belong to the same system and are situated in one or more Spaces. Each Context has a default Space, to which each agents belongs. In SARL, contexts do not have a state, they only serve as an organizational entity.

Spaces are defined as: *A Space is the support of the interaction between agents respecting the rules defined in a Space Specification. A Space Specification defines the rules (including action and perception) for interacting within a given set of Spaces respecting this specification.* In other words, a space defines how agents can interact with each other and with their environment. If an agent is situated in a space, he is assigned a unique address within that space. A specific type of Space is the EventSpace. An EventSpace allows its agents to trigger and react to events.

Spaces are accompanied by Space Specification. SpaceSpecifications define how a space should be instantiated and provide an implementation of the space. A space can then be created by invoking the getOrCreateSpaceWithSpec method on the current context.

Events

The basic construct for sending information to other agent is the Event. It is defined as: *An Event is the specification of some occurrence in a Space that may potentially trigger effects by a listener* [72]. Each event consists of a type or name, a source address, and optional data. In SARL, events are identified by a type, rather than a name.

Events can be sent within an event space in two different ways. They can be sent to all present agents in that space as follows:

```
emit( new Event( EventParameters ) );
```

An event can also be sent to a specific agent. In this case the address of the agent should also be specified:

```
emit( new Event( EventParameters ), DestinationAddress );
```

Agents and Behaviour

The core concept is the Agent. Agents are a member of a context and are situated in one or more of the context's spaces, where it can be contacted with its unique address.

As mentioned, when an agent is situated in an event-space, it can react to events that happen in that space. This is done with the special *on event [condition]{body}* statement. This statement allows programmers to define the agent's reactive behaviour.

An agent also has a set of skills, these skills define atomic actions the agent can perform in its space or context.

Capacities and Skills

As mentioned previously, an action is an atomic operation an agent can perform on itself or the environment. In SARL, actions are implemented by a method that can be called by an agent.

Methods that implement an agent's action are defined by Capacities. Capacities do not give an implementation for the actions, they only define what an agent can do, what behaviour is necessary for its execution. A Capacity is defined as: *A Capacity is the specification of a collection of actions* [72].

Implementations of a capacity are given by a Skill. Agent capacities are dynamic in that they can install new capacities during their execution, as well as change the associated Skill.

SARL also features a set of built-in capacities. These capacities define essential behaviour, such as creating and destroying agents, sending events, printing output, etc. Figure 3.9 shows a UML-diagram of the built-in capacities. We will now list their purpose:

- Schedules: Provides actions to schedule actions in the future. Actions can be scheduled once or can be repeated after a specified time-interval.
- ExternalContextAccess: Provides functionality to join or leave external contexts.
- DefaultContextInteractions: Provides agents with information about their default context, default space, and address. It also defines how agents can emit messages to all agents or a subset of the agents.
- InnerContextAccess: Defines how agents can access the inner context.
- Lifecycle: Defines actions to spawn new agents, and to kill an agent. SARL adheres to Fabien Michel's definition, in that only agents can create other agents [58]. Moreover, only the agent itself can decide wether it dies or not.
- Behaviour: Defines how agents can implement multiple behaviours, as well as choose which behaviour should be active. Note that we defined agents to have a single behaviour. In SARL the term behaviour is used to group a set of reactions. The collection of registered SARL-behaviours, together with the functionality that chooses which SARL-behaviour is active, defines the Agent's behaviour.
- Logging: Logging is not mentioned in Figure 3.9, however it is a built-in capacity that defines how agents can print messages to an output-stream. This is particularly useful for debugging or analysis.

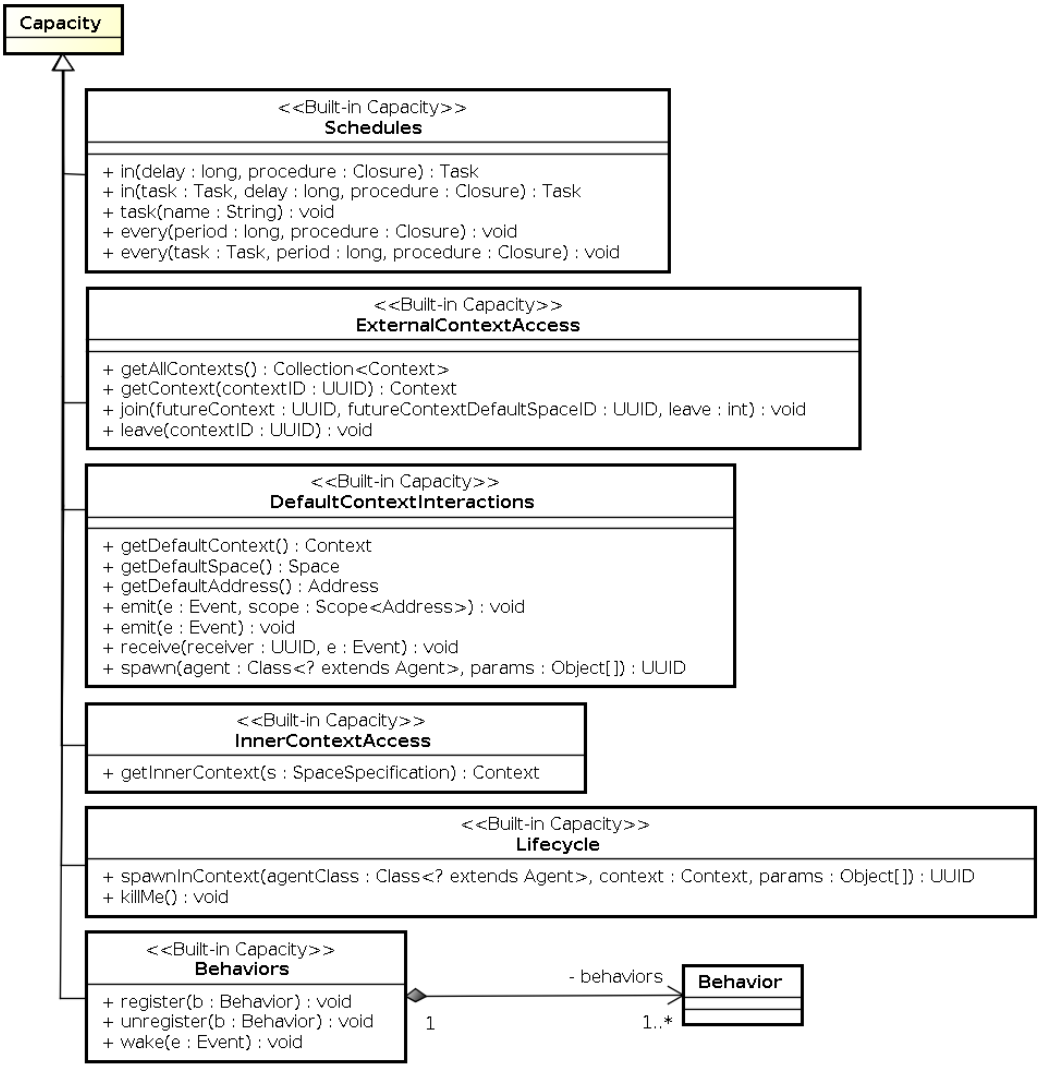


Figure 3.9: Built-in capacities of SARL [72]

CHAPTER 4

Design

In the previous chapter we analyzed the lack of formal semantics in current agent-based modeling and simulation tools. Based on existing formalisms, current tools, and the essential elements in an agent-based model, we will now propose a a generic formalism for agent-based modelling with precise semantics. Since the aim of this study is to analyze the benefits of implementing agent-based modelling tools based on precise semantics, we will demonstrate our claim by designing and implement such a tool. As discussed in the previous chapter, we will extend the SARL programming language, since it already features certain concepts as first class abstractions. It features autonomous agents, which can interact with each other in a space with events. Also, it features both reactive and pro-active behaviour.

In this chapter we will discuss the design goals of the tool, as well as the design. Finally, we will discuss how we will test the correctness of the tool as well as its usefulness.

4.1 Proposed formalism

In the previous sections, we analyzed the state-of-the art regarding formal semantics for agent-based modelling. We also studied the essential elements of an agent based model and we investigated how current ABM tools implement the agent-based modelling paradigm. In this section, we want to synthesize a precise semantics for agent-based modelling and simulation.

As we discussed previously, both the discrete event and the discrete time paradigm are valid choices for agent-based modelling and simulation. However, since the

discrete event paradigm allows models to schedule events at arbitrary time-steps, and since a discrete time model can be implemented by a discrete event model, we chose the discrete event paradigm for our formal semantics.

4.1.1 Requirements

Before we start designing our formalism, we need to define what we want to achieve with the formalism. These requirements will guide us to develop a useful formalism.

Since autonomy is a key feature of agent-hood, it needs to be represented adequately in the formalism. What we consider autonomy is that certain actions or behaviours of an agent can not directly be invoked by external entities (other agents or the environment). However, agents should be able to receive events or messages from the environment and the other agents, but it is the agent's own choice how to react to these external inputs.

Another key aspect we want to express extensively in our formalism is the interaction between the agent and its environment. The environment contains the global state of the model and directs how agents can communicate with each other. Even when agents set up a dedicated channel between them, messages are sent over a medium which is part of the environment.

Another important factor we want to take into account is reusability of artefacts. By providing a modular design, we want to encourage the reuse of agents or environments in different models.

The goal of this study is to provide a formalism that unifies the field of agent-based modelling and simulation. Therefore, we need a bare-bones algorithm that implements the essential features of an agent-based model and only those features. For example, organizational structures are a very important aspect in large heterogeneous agent populations [32]. In agent-based modelling and simulation, however, organizational constructs seem less important. Tools like NetLogo and GAMA, for example, do not have any built-in organizational structures. The reason for this is that agent-based modelling and simulation is often used for studying social interactions and situations. In these situations, agents often have no common goal, but a personal one and don't need complex organizations to achieve it. For this reason we will not include organizational structures as a first-class abstraction. On the other hand, we desire a formalism that is extensible. It should be relatively easy to extend our formalism such that it implements various specific aspects of agent-based modelling.

4.1.2 Influences and events

In our formal semantics, we want to base ourselves on the influence reaction model [56] to model the interactions of agents and the environment. The reason for this is

that the influence reaction model models real world interactions more convincingly than traditional action models.

To implement this model, we need to explicitly define influences. Influences are messages that can be sent to environment that contain information about the intentions of the sender. The environment can, in turn, send influences to the agents which contain information about the environment, or message, etc.

To represent influences, we base ourselves on the implementation of events in the SARL platform. In SARL, events have a type or name, a source address, and a optional data source. Thus, we define an influence as follows:

$$Influence = \langle \tau, src, \Theta \rangle$$

Where:

- τ is the type of the influence.
- src is the source address.
- Θ is a set of arbitrary values.

4.1.3 The agent-based model

Firstly, we will define the overcoupling model, called the agent-based model. In this model, we can define all agents and environments that exist in the model.

We define the agent-based model as follows:

$$ABM = \langle A_{abm}, AD_{abm}, E_{abm}, ED_{abm}, e, ZA, ZE \rangle$$

Where:

- A_{abm} is the set of agent identifiers.
- $AD_{abm} = \{AD_i | i \in A_{abm}\}$ is the set of agent definitions.
- E_{abm} is the set of environment identifiers.
- $ED_{abm} = \{ED_i | i \in E_{abm}\}$ is the set of environment definitions.
- $e \in E_{abm}$ is the default environment.
- $ZA = \{Z_{ij} | i \in A_{abm}, j \in E_{abm}\}$ with $Z_{ij} : I_{out,i} \rightarrow I_{in,j}$ the agent translation functions. $I_{out,i}$ is the set of all output influences of the agent with identifier i. $I_{in,j}$ is the set of input influences of the environment with identifier j.
- $ZE = \{Z_{ij} | i \in E_{abm}, j \in A_{abm}\}$ with $Z_{ij} : I_{out,i} \rightarrow I_{in,j}$ the environment translation functions. $I_{out,i}$ is the set of all output influences of the environment with identifier i. $I_{in,j}$ is the set of input influences of the agent with identifier j.

In our formalism, agents can not communicate with each other without contacting the environment first. To emphasize this, we created two separate translation functions. Translation functions are used to translate outgoing influences from agents and environments to incoming influences of environments and agents respectively.

One environment is reserved as the default environment and contains all agents that are alive. Whenever an agent is born, it is added to the participants of the default environment. Whenever an agent leaves the default environment, the agent is considered dead. A special purpose of this environment is to inform agents of environments they can join. Agents should not have full access to the global state, which means that they do not know what environments are out there. They discover joinable environments by querying environments in which the agent is situated. Since every living agent is part of the default environment, the default environment should point them towards environments the agent can join.

4.1.4 The agent model

Now we will define the agent. As discussed in section 3.1, an agent has an autonomous behaviour that is either reactive, proactive, or a combination of both. It should also be able to communicate with other agents. Finally, an agent is situated in one or more environments.

After studying existing formalisms for agent-based modelling, we concluded that the action model of LDEF [8] gives a good representation of agent behaviour. However, there are some parts missing before we can integrate it in our formal semantics. Firstly, LDEF does not support the IRM4S model. Each action is handled separately. We need to modify the output function such that it can send a bag of influences. To do this, we use Müller's argument that parallel DEVS are an elegant implementation of the IRM4S. Secondly, in our formalism agents are situated in one or more environment, so each agent needs to maintain a set of environment identifiers in which it is situated, as well as its address in each environment.

The agent is defined as follows:

$$Agent = \langle I_{in}, I_{out}, S_{aw}, S_{cond}, S_{act}, P, D_{react}, D_{proact}, A, ta \rangle$$

Where:

- I_{in} is the set of input influences the agent can receive from the environment.
- I_{out} is the set of output influences the agent can send to the environment.
- S_{aw} is the set of situation awareness states. These contain the agent's knowledge or beliefs about its environment and itself.
- S_{cond} is the set of decision condition states.
- S_{act} is the set of action states

- $P : I_{in}^{+} \times Q \rightarrow S_{aw}$ is the perception function, where $Q = \{(s, e) | s \in S_{aw}, 0 \leq e \leq ta(S_{act})\}$ and I_{in}^{+} is the set of all possible bags over I_{in}.
- $D_{react} : S_{aw} \times S_{cond} \times S_{act} \rightarrow S_{act} \times S_{cond}$ is the reaction decision function.
- $D_{proact} : S_{aw} \times S_{cond} \times S_{act} \rightarrow S_{act} \times S_{cond}$ is the proactive decision function.
- $A : S_{act} \rightarrow 2^{envs, I_{out}}$ is the action function, where $envs \subseteq E_{abm}$.
- $ta : S_{act} \rightarrow \mathbb{R}_{0,+\infty}^{+}$ is the time advance function.

The agent model is interpreted as follows. The state of an agent can be defined by $s = \{s_{aw}, s_{cond}, s_{act}\}$ where $s_{aw} \in S_{aw}, s_{cond} \in S_{cond}$, and $s_{act} \in S_{act}$. If an agent receives no external influences, it will remain in its state for a time $ta(s_{act})$. When $ta(s_{act})$ expires, the model will calculate its output influences by invoking $A(s_{act})$. Then the agent will perform its next state transition. Since no external influences were perceived, the next state is computed as $s' = \{s_{aw}, D_{proact}(s_{aw}, s_{cond}, s_{act})\}$. In this case, the agent exhibits pro-active behaviour. Even though no external influence is perceived, the agent still schedules transitions and performs actions to create opportunities.

When an external influence is perceived by the agent before the next scheduled transition, the agent will perform a state transition and the next state is given by $s' = \{s'_{aw} = P(\{i_1, i_2, ..., i_n\}, s_{aw}, e), D_{react}(s'_{aw}, s_{cond}, s_{act})\}$, where $i_1, i_2, ..., i_n \in I_{in}$. In this case the agent exhibits reactive behaviour, since it reacts to external events.

In both cases, the time the agent will remain in its new state without external influences is given by $ta(s_{act})$.

Our agent model is also compatible with the BDI model. The beliefs of the agent are stored in the situation awareness state. The beliefs can be updated with the perception function, which only perceives the influences given by the environment. This allows the environment to choose wether the agent receives global or partial information. The goals or desires are modelled in the S_{cond}, while the intentions are modelled in the decision and action functions.

We do not treat a collision between an internal and an external transition as a special case, like in parallel-DEVS. This is because in the case of a collision of an external and internal state transition, the agent will first perceive its environment. Since the transition function D is used in both cases, we see no need for a specific function for this special case. During such a collision the agent will first compute its output influences, then it will perceive its environment and then decide its next state.

4.1.5 The environment

Now we will define the environment. Here-for, we will not base ourselves on the LDEF implementation for environments. The reason for this is that the LDEF over complicates the implementation of environments, which in turn, obfuscates the

semantics of the environment. In LDEF an environment is comprised out of multiple environment element models, which are coupled into environment models and multi-environment models. However, from the point of view of the agents, the environment is a single interface to which agents send their influences and which provides them with information. We therefore chose to implement each environment as a single model. If the environment comprises multiple entities, their individual states can be aggregated into the environment state. Parunak [66] formalizes the environment as a tuple $\langle State_e, Process_e \rangle$, where $Process_e$ defines how the state changes over time. However, to implement to IRM4S, we need to extend Parunak's formalization such that it can receive, send, and react to influences. Therefore, we extended the atomic model of the parallel DEVS formalism, such that it implements our needs.

We define the environment as follows:

$$Environment = \langle I_{in}, I_{out}, S_{env}, A, \delta_{env}, \delta_{react}, \delta_{conf} \lambda_{env}, ta \rangle$$

Where:

- I_{in} is the set of input influences.
- I_{out} is the set of output influences.
- S_{env} is the set of environment states.
- A is the set of agent identifiers.
- $\delta_{env} : S_{env} \to S_{env}$ is the environment state transition function.
- $\delta_{react} : Q \times I_{in}^{+} \to S_{env} \times 2^{A_{abm}}$ is the reaction function. Here is I_{in}^{+} the set of all possible bags over I_{in}.
- $\delta_{conf} : S_{env} \times I_{in}^{+} \to S_{env} \times 2^{A_{abm}}$ is the confluence function.
- $\lambda_{env} : S_{env} \to 2^{I_{out} \times A}$ the output function.
- $ta : S_{env} \to \mathbb{R}$ is the time-advance function.

The model is then interpreted as follows. At each point in time, the environment is in a state $s \in S_{env}$ and has a set of agents situated in it, given by A. If no influences are received from agents, it will remain in its state for a time $ta(s)$. When $ta(s)$ expires, the output influences of the environment are computed by $\lambda_{env}(s)$. Note that each output element of λ_{env} is a tuple containing an influence and the identifier of the agent that should receive it. Then the environment computes its new state and agent set with $\delta_{env}(s)$.

If a set of influences $I_{received} \subseteq I_{in}^{+}$ is received before $ta(s)$ expires, the environment should react to these influences. This is done by invoking $\delta_{react}(s, e, I_{received})$, where e is the elapsed time between the last state transition and the reception of $I_{received}$. This method returns the new state and agent set of the environment and thus defines how the environment will react. In both cases, the environment will remain in the new state s' for a time $ta(s')$.

Environment completeness

Russel and Norvig [74] discuss several properties of the environment: *accessibility, dynamism, determinism, and continuity.* We already discussed how the modeller can choose how much information is passed to the agents from the environment. As for dynamism, we can use δ_{env} and the time-advance function to perform environment specific transitions. If we let the time-advance function return $+\infty$ for every input value, the state of the environment will only change due to influences of the agent, thus defining a static environment.

Because we implement the IRM4S, we specialize the formalism to implement non-deterministic environments. However, if the reaction transition function is implemented such that each influence has a predefined result, deterministic environments can be implemented.

Our formalism implements the discrete event paradigm. This means that the state of both the environment and the agent are discrete, but can change on arbitrary time steps. It is currently infeasible to implement an environment with a state variable that changes continuously through time. However, we can compute the next state transition based on a continuous function.

4.1.6 Communication

Note that we have not defined communication specific semantics and, in our formalism, agents can only communicate with the environment. This is done deliberately to allow the modelling of faulty communication between agents. In real world systems, messages have to pass through the environment in some way. They are sent as signals over a medium, or by vocal or visual signs. Certain conditions in the environment can cause complications. For example, a strong magnetic field can disrupt a wifi signal. To allow modellers to incorporate these interactions, messages need to be sent as an influence to the environment. The environment will then decide how the message is sent to the receiver.

Because communication is such an important aspects of agents, we will define messages as a special type of influences. A message is defined as:

$$Message = \langle Message, Src, \{Dst, Value\} \rangle$$

Where:

- $Message$ is the type of the influence.
- Src is the address of the entity that sends the message.
- Dst is the address of the destination agent within the environment.
- $Value$ is the content of the message.

4.1.7 Dynamic structure

As discussed previously, the dynamic structure is an inherent feature of an agent-based model. However, the structure of the model can only change in a few predefined ways. A new agent can be born, which increases the overall population, and agents can die which decreases the population. On top of that, agents can also join and leave an environment.

In our formalism, environments can change the agent set that is present in the environment as a reaction to influences. So to allow dynamic structure, we just need to define the influences that can be sent by the agents. If an agent is not in any environment, it can be considered death. It can request itself to join and leave an environment, and it can request to spawn another agent in the environment. These influences are defined as follows:

$$Join = \langle Join, Src, \phi \rangle$$

$$Leave = \langle Leave, Src, \phi \rangle$$

$$Spawn = \langle Spawn, Src, a \rangle \text{ where } a \in A_{abm}$$

By defining them as influences, it is up to the modeller to define how and if the agents can join the environment. For example, if an environment has a certain maximum capacity, it can deny the join of new agents if it is full.

4.1.8 Precise semantics

In the previous sections we informally described the semantics of each of the components of the formalism. However, one of the requirements of our formalism is that it has precise semantics. With precise we meant that we describe the semantics in such a way that anyone can implement them only by analyzing this document. However, we did not use a formalization language to denote our semantics.

First we will define the accessible fields of agent models and environments. These can also be found back in the definitions, but Table 4.1 gives an overview of the representations in the following pseudocode.

First we start by defining two methods that are used to schedule entities, as well as recieve the next step in the schedule that needs to be simulated.

Algorithm 4.1: The scheduleTransition method (pseudo code)

```
1  name: scheduleTransition
2  input: schedule, id, timestep
3  output: updatedSchedule
4  begin
5      updatedSchedule ← schedule
6      //remove all existing scheduled transitions with id
7      for scheduled in {(time, id)|(time, id) ∈ schedule}:
8          updatedSchedule ← updatedSchedule \ scheduled
9      // add new entry to the schedule
```

Table 4.1: Overview of accessible fields of entities in the simulation

name	type	description
	Agent Model Fields	
$s_{aw} \in S_{aw}$	state	The current awareness state of the agent
$s_{cond} \in S_{cond}$	state	The current condition state of the agent
$s_{act} \in S_{act}$	state	The current action state of the agent
$P(2^{I_{in}}, S_{aw}, Number)$	method	The perceive function
$D_{react}(S_{aw}, S_{cond}, S_{act}) : S_{act}$	method	The reaction decision method
$D_{proact}(S_{aw}, S_{cond}, S_{act}) : S_{act}$	method	The proactive decision method
$A(S_{act}) : 2^{Id \times I_{out}}$	method	The act method
$ta(S_{act}) : Number$	method	The time advance function
	Environment Model Fields	
$s_{env} \in S_{env}$	state	The current state of the environment
$participants \in A$	state	The current set of participants in the environment
$delta_{env}(S_{env}) : S_{env}$	method	The environment state transition function
$delta_{react}(2^{I_{in}}, S_{aw}, Number) : S_{env}$	method	The reaction function
$delta_{conf}(2^{I_{in}}, S_{aw}) : S_{env}$	method	The confluence transition function
$ta(S_{env}) : Number$	method	The time advance function
	Influence Fields	
source	identifier	the identifier of the sender of the influence

```
10      updatedSchedule ← updatedSchedule ∪ {(timestep, id)}
11      return updatedSchedule
12  end
```

Algorithm 4.2: getNextTimeStep method (pseudo code)

```
1   input: schedule
2   output: nextTimeStep, updatedSchedule
3   begin
4       timestep ← min{k|(k, x) ∈ schedule}
5       scheduledIDs ← {id|(time, id) ∈ schedule}
6       nextTimeStep ← (timestep, scheduledIDs)
7       updatedSchedule ← schedule \ {(k, v)|k = timestep, (k, v) ∈ schedule}
8       return nextTimeStep, updatedSchedule
9   end
```

Algorithm 4.3: Simulation algorithm (pseudo code)

```
1   schedule ← ∅
2   agentSet ← {(i, am_i)|i ∈ A_abm, am_i ∈ AD_abm}
3   environmentSet ← {(i, em_i)|i ∈ E_abm, em_i ∈ ED_abm}
4   time ← 0
5   lastTransitions ← {(entity, 0)|(id, entity) ∈ agentSet ∨ (id, entity) ∈ environmentSet}
6
7   for (id, agentmodel) in agentSet:
8       schedule ← scheduleTransition(schedule, id, agentmodel.ta(agentmodel.s_a))
9   for (id, environmentmodel) in environmentset:
10      schedule ← scheduleTransition(schedule, id, environmentmodel.ta(environmentmodel.s_env))
11
12  while schedule ≠ ∅:
```

```
13      (s_time, ids), schedule ← getNextTimeStep(schedule)
14      time ← s_time
15
16      scheduledAgents ← {agent|(id, agent) ∈ agentSet ∧ id ∈ ids}
17      scheduledEnvironments ← {env|(id, env) ∈ environmentSet ∧ id ∈ ids}
18      influencedAgents ← ∅
19      influencedEnvironments ← ∅
20      influenceMap ← ∅
21
22      for agent in scheduledAgents:
23          for (id, influence) in agent.act(agent.s_a):
24              iEnv ← environment with (id, environment) ∈ environmentSet
25              influencedEnvironments ← influencedEnvironments ∪ {iEnv}
26              influenceMap ← influenceMap ∪ {(id, influence)}
27
28      for environment in scheduledEnvironments:
29          for (id, influence) in environment.lambda(environment.s):
30              iAgent ← agent with (id, agent) ∈ agentSet
31              influencedAgents ← influencedAgents ∪ {iAgent}
32              influenceMap ← influenceMap ∪ {(id, influence)}
33
34
35      for agent in influencedAgents ∪ scheduledAgents:
36          agentId ← id with (id, agent) ∈ agentSet
37          lastTransition ← t with (agent, t) ∈ lastTransitions
38          lastTransitions ← (lastTransitions \ (agent, lastTransition)) ∪ (agent, time)
39          if agent ∈ influencedAgents:
40              influences ← {Z_{environmentID,agentID}(influence)|
41                    (agentId, influence) ∈ influenceMap ∧ environmentID = influence.source}
42              agent.s_aw ← agent.P(influences, (agent.s_aw, time − lastTransition))
43              agent.s_cond, agent.s_act ← agent.D_react(agent.s_aw, agent.s_cond, agent.s_act)
44          else:
45              agent.s_cond, agent.s_act ← agent.D_proact(agent.s_aw, agent.s_cond, agent.s_act)
46          schedule(id, time + agent.ta(agent.s_act))
47
48      for environment in influencedEnvironments ∪ scheduledEnvironments:
49          evnIdid with (id, environment) ∈ agentSet
50          lastTransition ← t with (environment, t) ∈ lastTransitions
51          lastTransitions ← (lastTransitions \ (environment, lastTransition)) ∪ (environment, time)
52
53          if environment ∈ scheduledEnvironments \ influencedEnvironments:
54              environment.s_env ← environment.delta_int(environment.s_env)
55          if environment ∈ influencedEnvironments \ scheduledEnvironments:
56              influences ← {Z_{agentID,envID}(influence)|
57                        (envId, influence) ∈ influenceMap ∧ agentID = influence.source}
58              environment.s_env ← environment.delta_int(influences, environment.s_env, time − lastTransition)
59          if environment ∈ scheduledEnvironments ∩ influencedEnvironments:
60              influences ← {Z_{agentID,envID}(influence)|
61                        (envId, influence) ∈ influenceMap ∧ agentID = influence.source}
62              environment.s_env ← environment.delta_conf(influences, environment.s_env)
63          schedule(id, time + environment.ta(environment.s_env))
```

4.2 Extending the SARL meta-model

Currently, SARL does not feature any simulation capabilities. However, one of the design goals of our simulation platform is to represent concepts of agent-based modeling and simulation as close as possible to the theoretical definition. Since SARL features many of the concepts from section 3.1 as first class abstractions: agents,

behaviour, communication. We aim to extend the SARL meta-model, such that it includes functionality for modelling & simulation which implements our proposed formal semantics.

Figure 4.1 shows the extended meta-model. We updated the existing SARL meta-model with artefacts that will allow us to implement

4.2.1 The simulator and virtual clock

To be able to run simulations, we need a simulator. The simulator keeps track of the simulation time and the schedule of the upcoming state transitions. The simulator is in charge of invoking each of the agents at the right time and that influences are delivered to the correct destination.

4.2.2 Simulation environments and influences

As discussed in our proposed formal semantics, we need to define influences as a first class abstraction. Since influences contain the same information as a SARL Event, we can add influences as a specific type (or sub-type) of events.

We also defined SimulationEnvironment as a sub-type of EventSpace. The SimulationEnvironment will be used to explicitly model the environment in which the simulation takes place. Within these environments, influences can be sent to agents and agents can send influences to the environment.

4.2.3 SimAgents

The SimAgent is a sub-type of agent that can be used in simulations. These agents will be able to send and receive influences, as well as define the behaviour of the agent in this situation.

Currently a lot of functionality is given by the built-in capacities. However, these built-in capacities are created to develop multi-agent systems. We therefore add specific simulation skills that will implement these capacities for modelling and simulation.

4.3 Implementation

In this section, we will discuss the implementation choices made to develop the simulation platform, based on the SARL programming language.

The goal of this tool is to implement the formalism with precise semantics described in section 4.1. Additionally, we want to make sure that artefacts in a model (agent-models, environments, ...) can be reused in other models.

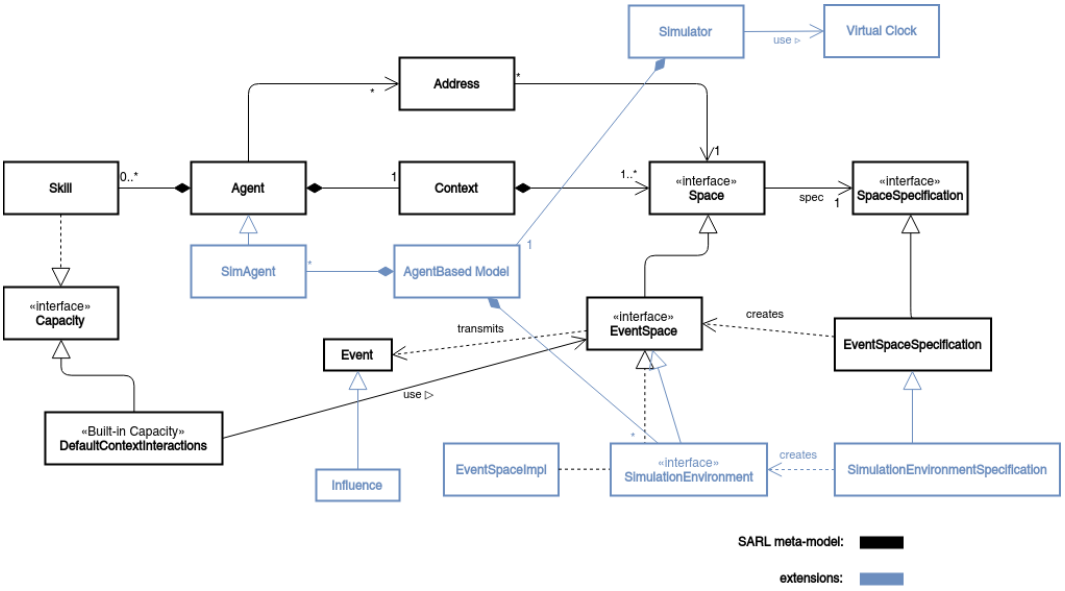


Figure 4.1: The extended meta-model

4.3.1 The simulation runtime environment

In the current execution of SARL programs, SARL Agents are deployed on the Janus platform. The Janus platform handles the instantiation of the Agent Context, as well as the default space in which all agents within the same context are situated. The Janus platform is developed for creating multi-agent based applications, either distributed or on a single device.

To implement the simulation platform, we were left with two options. Firstly, we could use SARL's current deployment approach and use SARL artefacts to implement the simulator. An evident design would be to implement the simulator as an agent, which will maintain a schedule, the virtual clock, and will direct all agents involved in the simulation. This approach has been used, for example, in the development of TurtleKit [57], which is based on the MadKit multi-agent platform [39].

Secondly, we could provide a different runtime environment, in which agents can be simulated. This environment will take up the role of maintaining the agent context and spaces, as well as directing all agents within the simulation. This runtime environment will take over the responsibilities of Janus that apply to a modelling and simulation context.

In the first option, we reuse the complete operational semantics of SARL and the Janus platform. Although this is beneficial for reuse of artefacts, the operational semantics of Janus are specialised for deploying multi-agent systems. One such example is how *on* statements are handled in Janus. For each *on* statement in each agent a threat is launched that handles the reaction of the agent. Since our aim

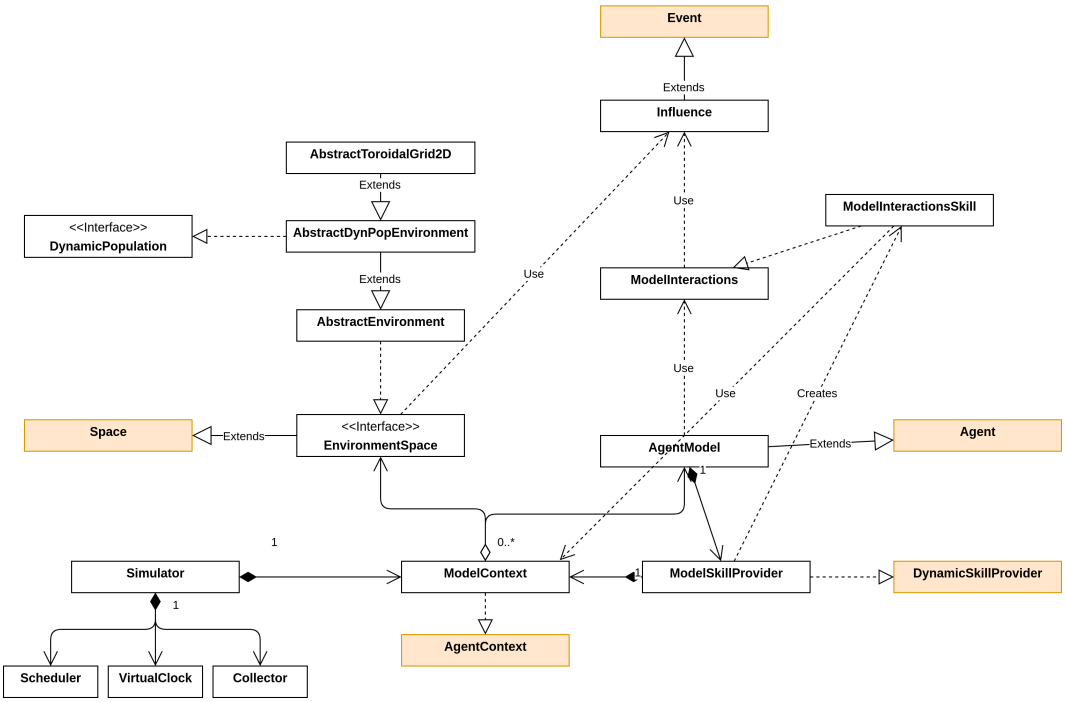


Figure 4.2: The class diagram of SARLforSIM

is to develop a simulation platform to run on a single machine, we do not have access to a lot of threads. Moreover, the added overhead needed to synchronize the simulation at each timestep will probably outweigh the gain in efficiency by the multi-threading. The synchronization is needed because we cannot start the next simulation step before the previous one is finished. Since our simulation algorithm comprises two phases (influence phase and reaction phase), this overhead is doubled. Therefore, we opted for the second option.

A UML diagram of the platform can be seen in Figure 4.2. We will now discuss each of the components in more detail.

The ModelContext class

As seen in the SARL meta-model, all agents are contained within a context. The SARL interface that defines this context is the AgentContext interface. For our model and simulation platform, we implemented this interface in the ModelContext class.

The main purpose of this entity is to create and manage EnvironmentSpaces and AgentModels. During a simulation, this entity will be contacted by the environments to create or delete agents. The simulator uses this entity to retrieve the current set of EnvironmentSpaces and AgentModels.

The ModelInteractions capacity and skill

To let agents interact within a ModelContext, we created the ModelInteraction capacity and skill. The capacity comprises different ways of sending influences. These ways include:

- Sending an influence to a single entity.
- Broadcasting an influence to all environments in which an agent is situated.
- Broadcasting an influence to all agents within the environment.
- Schedule next next transition.

With the capacity, we implemented a corresponding skill, the modelInteractionsSkill. Internally, the skill makes use of the provided model context to send messages to the appropriate entities. However, this is completely hided from the developer, since the ModelInteractions capacity is a built-in capacity and its implementation is given by a dynamic skill provider, see Section 4.3.1.

The SimSkillProvider class

SARL agents feature a dynamic skill provider. This provides the appropriate skill for the built-in capacities, without manually installing skills. For our platform, it was useful to implement a dynamic skill provider for modelling and simulation. The benefits are twofold. First, we can determine which capacities are built-in. Since scheduling is handled by the time-advance function, there is no need for providing the scheduling capacity to modelled agents. Secondly, we can provide the skills that are specialized for modelling and simulation.

One example is the Logging capacity. We provided a skill of this capacity that is specifically made for the modelling and simulation context. The SimSkillProvider will provide this implementation over the standard implementation of SARL.

The AgentModel class

The AgentModel is an abstract agent, that provides the necessary functionality to implement the behaviour of an agent model. Since this implementation implements the agent model of our formalism, it should implement the perception function, the decision function, and the act function. To model a custom agent model, modellers can extend the AgentModel and implement the abstract functions.

Listing 4.4 shows the implementation of the AgentModel. The AgentModel provides a skeleton of all functions that need to be implemented by the user. In contrast to the agent model in the proposed formalism, the SARLforSIM AgentModel does not have a distinction between the awareness state, the condition state, and the action state. The complete state of the object comprises the three state aspects of the original

agent model of the formalism. Dedicated states where not developed because the implementation of state can become very complex. To keep the structure of the agent model simple, we decided to omit the three different states. However, it is advisable to update only the appropriate fields in each method.

The AgentModel features the same functions as the original agent model in the proposed formalism. Each of the methods is invoked by the corresponding *on* statement. The on statements are used to react to events send by the simulator. By using the *on* statements, we aim to increase the reusability of artefacts. When the agents are deployed on the Janus platform, rather than our simulation platform, we can make use of self influences to direct the behaviour of the agents.

By reusing the SARL agent class, we make sure that agents are indeed autonomous. SARL's syntax allows agents to only define private or protected methods, which means that, when the user defines a different package, only the instance of the agent itself has access to its methods. An agent can only receive and react to events, which are sent by the simulator.

Listing 4.4: AgentModel code

```
1  abstract agent AgentModel {
2          uses Logging, ModelInteractions
3
4          protected var collector : Collector = null
5
6          on Initialize {
7                  this.collector = occurrence.parameters.get(0) as Collector
8          }
9
10         on Destroy {
11                 // Event trigger when the agent is destroyed from the system.
12                 info("The agent was stopped.")
13         }
14
15         on Perceive {
16                 // Event trigger when an influence is received
17                 info("Received influences")
18         }
19
20         on Act {
21                 act()
22         }
23
24         on Decide {
25                 decide(occurrence.perceived)
26         }
27
28         on Schedule {
29                 scheduleTransition(timeAdvance);
30         }
31
32         def decide(percieved : boolean) : void
33
34         def act():void
35
36         def timeAdvance():double
37
38 }
```

The EnvironmentSpace interface

The closest artefact in SARL to an environment is a Space. Spaces define how agents can interact with each other within a single context. The space is simple interface that defines that agents have a set of participants and a dedicated IP address. As seen in our proposed formalism, an environment maintains a state and has a set of functions that define its behaviour. We extended the existing interface, such that it provides a blue print for any user defined EnvironmentSpace, as seen in Figure 4.3.

The EnvironmentSpace feature a few methods that are not related to the environment model in our formalism. This is due to our mechanism of sending events from agents to environments. Because agents only have private methods, we cannot just simply collect their output directly. To solve this problem we considered three options:

- The first option is to keep an influence buffer in each Environment. When an agent sends one or more influences, they are stored in the influence buffer of the destination environment. Whenever the environment needs to perform a reaction, it can look at its internal buffer for the set of influences.
- The second option is to maintain an influence buffer in the simulator. This buffer is passed by reference along with the perceive event . Then the agent can use that buffer to store its influences. This might become a problem in a distributed simulation tool, where only a copy of the buffer can be sent.
- The third option is to define a subclass of Agent, which has public methods. This is feasible since SARL and Java are completely inter-operable. However, this violates the autonomy property of agents.

For the first option, the simulator becomes more intertwined with the model entities. Ideally, the simulator only collects data from the model entities and redistributes it. However, for the first option, the environment maintains information that should be kept in the simulator. Although this option adds more coupling, we still think it is the most appropriate option. Agents send influences to the environments, but since the reaction phase comes later in the simulation process, the environments need to buffer them until that phase is started. Figure 4.4 shows a sequence diagram of this mechanism. The second option is a workaround that does not work under any circumstances. If our tool is extended such that it allows distributed simulation, we cannot pass arguments by reference. The third option violates the autonomy property of agents. Since this is a key feature we want to show in our platform, we cannot choose this option.

Since these functions are not part of the model definition, their implementation is not the developer's responsibility. Therefor, we created the AbstractEnvironment class, which provides a default implementation for each of those functions.

Adding or removing agents is handled by environments in our formalism. However, it is the ModelContext that maintains a list of all agents. To notify the ModelContext

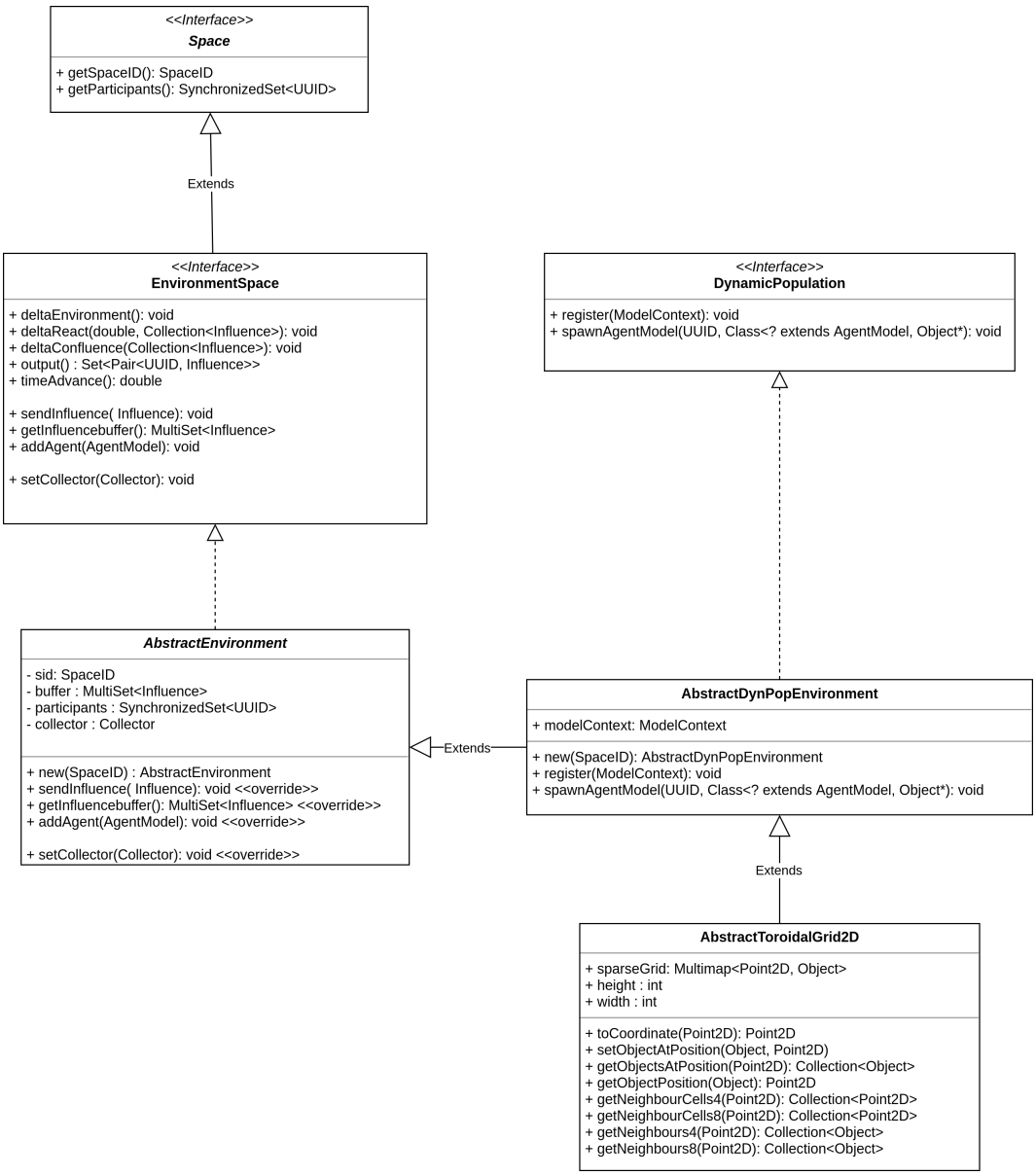


Figure 4.3: UML diagram of the EnvironmentSpace in SARL

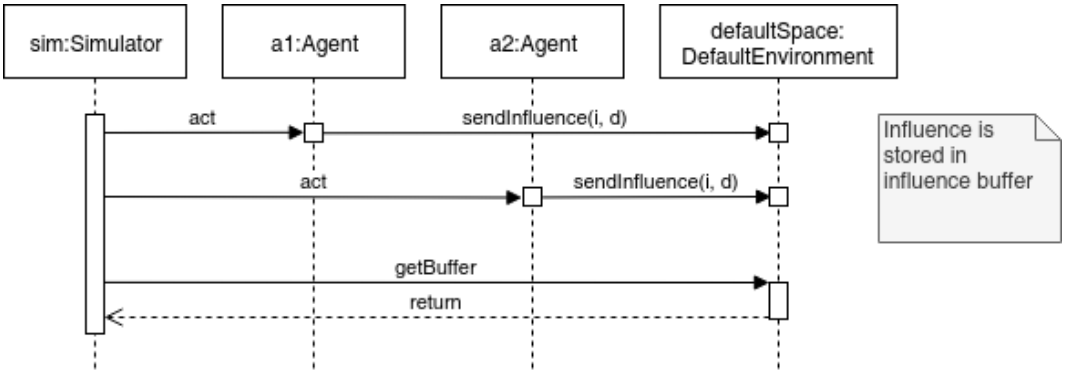


Figure 4.4: Sequence diagram of how agents send their influences to the environments

of such changes, we used the observer pattern. In the interface DynamicPopulation, we provide a method to register a ModelContext and a method to spawn an agent in the current environment. Again, we provide a default implementation for Environments with a dynamic population, called the AbstractDynPopEnvironment.

Agents are often situated in a physical spatial environment in which they can move freely. Our tool provides an example of such a spatial environment, the AbstractToroidalGrid2D. This class provides a grid in which objects can be placed and moved. When agents move over the edge of the grid, they move to the cell on the opposite side of the grid. We chose this type of environment, since it is the default environment in NetLogo.

The Simulator class

The final piece of the puzzle is the Simulator. The simulator maintains the ModelContext as well as the virtual clock and the schedule. This schedule is a macro-level schedule and keeps track of which entity will perform a state transition.

The simulator will simulate timesteps, according to the algorithm in Listing 4.3 until a stop condition is met.

For the simulator, we allow three different stop conditions. The first stop condition is when no more scheduled transitions are left. In this case we computed the end-state of the simulation prematurely and can stop any further computations.

The second stop condition is the stop time, which can be set by the modeller. Whenever the simulation time exceeds the stop time, any further computations are cancelled and the simulation stops.

The final option is to define a lambda function that returns true if the simulation should be stopped, and false otherwise. This lambda function gets the simulator

itself as parameter, such that the complete state of the model and simulator can be inspected by the test function.

The EventDispatcher class

The EventDispatcher allows the simulator entity to direct the agents by sending events. A very important purpose of the EventDispatcher is to execute the agents in sequence. This allows us to eliminate any synchronization that would be needed during each of the simulation phases. In turn this allows us to implement the simulator more efficiently.

Listing 4.5: The sendEvent function

```
1  def sendEvent(^event : Event, ^agent : AgentModel) {
2                  var handlers : Collection<Runnable> = new ArrayList
3                  evaluatorRegistry.register(^agent)
4                  var evaluators = evaluatorRegistry.getBehaviourGuardEvaluatorsFor(^event,
5                  for (evaluator : evaluators) {
6                          evaluator.evaluateGuard(^event, handlers)
7                  }
8                  for (handler : handlers) {
9                          handler.run
10                 }
11         }
```

The Collector interface

The final entity we discuss in more detail is the Collector. The Collector is used to store data during the simulation. Each entity in the simulation has access to the Collector and can add entries. When a simulation is done, we can retrieve the collector and process the data. The current collector that is provided by default only allows for string entries, but more complex collectors can be defined and added.

CHAPTER 5

Tool Evaluation

In the previous chapter, we described the design of the agent-based modelling and simulation platform SARLforSIM. To answer the research questions in Chapter 1, we designed a number of experiments to evaluate the platform. In this section, we will discuss in detail the performed experiments as well as their results.

5.1 Test case

The goal of this study is to propose a tool that implements precise semantics. To test this, we implemented a test suite that checks whether the output traces of models in our proposed formalism are identical to the output traces in their implementation in SARLforSIM.

The reason for this is twofold. Firstly, these scenarios allow us to show the correctness of the implementation of the formalism in the SARLforSIM platform. We will compute the trace of the simulation by hand according to our formalism and compare it with the trace provided by platform. Secondly, this allows us to show how common practices in agent-based modelling can be performed in our platform.

5.1.1 Reactive environment - traffic light switcher

The first scenario is a simple scenario that demonstrates how agents can interrupt their environment at arbitrary time steps. In this scenario, the environment consists of a traffic light which is operated by an agent. The environment has three states: *Light is red*, *Light is green*, and *Light is orange*. The environment can react to three

possible input events: *Set light to green*, *Set light to red*, and *Set light to orange*. The environment does not respond to the agent (the output function always returns the empty set), and the environment can be considered deterministic, since only a single agent will act upon the environment.

The agent will operate the traffic light by sending influences at specific time steps. The agents goal is to operate the light such that the light follows the diagram shown in Figure 5.1.

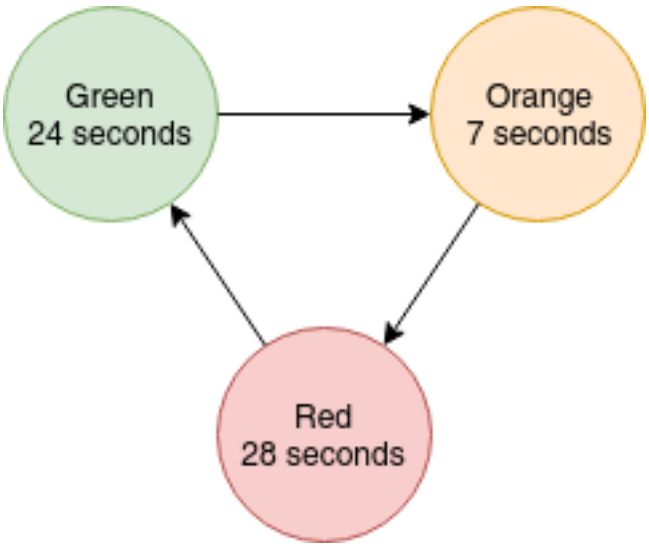


Figure 5.1: Diagram of the traffic light

For this simple model we provide a more formal representation of the model in our formalism:

$$ABM = \langle A, AD, E, ED \rangle$$

With:

- $A = \{AgentID\}$
- $AD = \{Agent\}$
- $E = \{EnvID\}$
- $ED = \{Environment\}$

$$Agent = \langle I_{in}, I_{out}, S_{aw}, S_{cond}, S_{act}, P, D, A, ta \rangle$$

with:

- $I_{in} = \emptyset$
- $I_{out} = \{$*Set light to green*, *Set light to red*, *Set light to orange*$\}$
- $S_{aw} = \emptyset$ We don't get any input, so we don't perceive anything.

- $S_{cond} = \{Light\ is\ red,\ Light\ is\ green,\ Light\ is\ orange\}$
- $S_{act} = \{To\ green,\ To\ red,\ To\ orange\}$
- $P =$ not applicable
- $D = \{(Light\ is\ red \rightarrow \{S_{cond} = Light\ is\ green, S_a = To\ green\}),$
  $(Light\ is\ green \rightarrow \{S_{cond} = Light\ is\ orange, S_a = To\ orange\}),$
  $(Light\ is\ orange \rightarrow \{S_{cond} = Light\ is\ red, S_a = To\ red\})\}$
- $A = \{(To\ green \rightarrow \{Set\ light\ to\ green, EnvID\}),$
  $(To\ orange \rightarrow \{Set\ light\ to\ orange, EnvID\}),$
  $(To\ green \rightarrow \{Set\ light\ to\ red, EnvID\})\}$
- $ta = \{(To\ green \rightarrow 28), To\ red \rightarrow 7), (To\ orange \rightarrow 24)\}$

$$Environment = \langle I_{in}, I_{out}, S_{env}, \alpha, \delta_{env}, \delta_{react}, \delta_{conf}, \lambda_{env}, ta \rangle$$

with:

- $I_{in} = \{Set\ light\ to\ green,\ Set\ light\ to\ red,\ Set\ light\ to\ orange\}$
- $I_{out} = \emptyset$
- $S_{env} = \{Light\ is\ red,\ Light\ is\ green,\ Light\ is\ orange\}$
- $\alpha = \{AgentID\}$
- $\delta_{env} = \{(Light\ is\ red \rightarrow Light\ is\ red),(Light\ is\ green \rightarrow Light\ is\ green),$
  $(Light\ is\ orange \rightarrow Light\ is\ orange)\}$
- $\delta_{react} = \{(\textbf{on}\ Set\ light\ to\ red \rightarrow Light\ is\ red \times \{AgentID\}),$
  $(\textbf{on}\ Set\ light\ to\ green \rightarrow Light\ is\ green \times \{AgentID\}),$
  $(\textbf{on}\ Set\ light\ to\ orange \rightarrow Light\ is\ orange \times \{AgentID\})\}$ [1]
- $\delta_{con} = \delta_{env}$
- $\lambda_{env} = \emptyset$
- $ta = +\infty$ for any input

To study this model, we need to compute an output trace, based on the precise semantics of our proposed formalism. Since it is a relatively simple model, creating a trace is quite trivial. If the systems starts with the light on Green and we inspect only the state changes of the environment, we expect a trace as seen in Table 5.1:

We implemented this model in our platform as follows: Since the time advance of the environment is always $+\infty$, we only need to take care of the reaction function, as seen in Listing 5.1.

[1] We wrote "**on** *Event* $\rightarrow$ " to denote that the environment will react this way regardless of elapsed time or other received influences.

Table 5.1: Expected output trace

time step	Light color	Elapsed Time
0	Green	0
24	Orange	24
31	Red	7
59	Green	28
83	Orange	24
90	Red	7
118	Green	28
...	...	...

Listing 5.1: reaction function of the traffic light environment code

```
1  def delta_react(elapsed_time : double,
2                  influence_bag : Collection<Influence>)
3  {
4    for(influence : influence_bag){
5      if(influence instanceof LightToGreen) {
6        CurrentColor = Color.Green
7          }
8          if (influence instanceof LightToOrange) {
9            CurrentColor = Color.Orange
10         }
11         if(influence instanceof LightToRed) {
12           CurrentColor = Color.Red
13         }
14   }
15   System.out.println("Elapsed time: " + elapsed_time)
16   System.out.println("Current color: " + this.CurrentColor)
17 }
```

After running the simulation, we achieved following output trace:

```
Simulating Timestep 24.0
Elapsed time: 24.0
Current color: Orange
```

```
Simulating Timestep 31.0
Elapsed time: 7.0
Current color: Red
```

```
Simulating Timestep 59.0
Elapsed time: 28.0
Current color: Green
```

```
Simulating Timestep 83.0
Elapsed time: 24.0
Current color: Orange
```

```
Simulating Timestep 90.0
Elapsed time: 7.0
Current color: Red
```

```
Simulating Timestep 118.0
Elapsed time: 28.0
Current color: Green
```

Here we see that the output trace of the implemented model is equivalent to the expected output trace.

An important feature of our platform is the support of the influence reaction model by reusing the concepts of Parallel-DEVS. To show this functionallity, we update the previous model by adding another agent. The second agent, however, starts at a different phase of the light cycle. The environment should now be able to handle multiple inputs at the same time step. The difference between the previous model is that we use the rock-paper-scissor principle in the reaction function of the environment. If multiple influences to change the light are handled by the environment, Red will have precedence over orange and green, and orange has precedence over green. If agent 1 starts in the green phase, and agent 2 in the orange phase, this results in expected output trace in Table 5.2. The actual output trace of the model

Table 5.2: Expected output trace

time step	Influence Agent1	Influence Agent2	Environment Color
0	-	-	Green
7	-	ToRed	Red
24	ToOrange	-	Orange
31	ToRed	-	Red
35	-	ToGreen	Green
59	ToGreen	ToOrange	Orange
66	-	ToRed	Red
83	ToOrange	-	Orange
90	ToRed	-	Red
94	-	ToGreen	Green
118	ToGreen	ToOrange	Orange
...	...	...	...

is listed in Appendix A.1. The trace is identical to the expected output in Table 5.2.

5.1.2 Reactive agents - ping pong agents

This scenario shows how reactive agents can be created in SARLforSIM. For this we use the same example as the SARL introductory tutorial of the ping pong agents. In this model there are two agents, a ping and a pong agent. The ping agent will broadcast a ping event on receiving a pong event, while the pong agent broadcasts a pong event on a ping event. The ping agent will initialize the correspondence by sending the first ping event on initialization. The ping pong scenario shows an example of tropistic (or purely reacting) agents that act in a "tuple space"-like environment.

In our proposed formalism, the ping agent is encoded as follows:

$$PingAgent = \langle I_{in}, I_{out}, S_{aw}, S_{cond}, S_{act}, P, D, A, ta \rangle$$

Where:

- $I_{in} = \{PongEvent\}$
- $I_{out} = \{PingEvent\}$
- $S_{aw} = \{PongReceived, NoPongReceived\}$
- $S_{cond} = \emptyset$
- $S_{act} = \{SendPing, Idle\}$

The awareness state remembers the last received pong event. Since only two agents are present in the first version we don't have to take multiple simultaneous events into account. In the awareness state, we remember if the last received event was a Pong event or not. Thus we define P as follows: If there is a pong event in the set of input influences, set S_{aw} to PongReceived, otherwise to NoPongReceived.

We don't need any more deliberation, so we omit the conditional state. The decide function is implemented as follows: If S_{aw} is ReceivedPong, set S_{act} to SendPing, otherwise to Idle. The agent will then wait for 1 time unit before sending the Ping influence.

To inspect the output trace, we let the implemented agent output when he receives a ping, and when he sends pong (or vice versa). The expected output trace can be seen in Table 5.3.

The output trace of the scenario can be consulted in Appendix A.2. Note that for eacht timestep, two entries are present in the output trace. This is because the environment cannot perceive an influence and send it to its participants in the same timestep. When receiving a Ping event, the environment will schedule a next transition with time advance 0. In the next simulation step, the environment will send the influence to all its participants. Although these are two seperate simulation steps, they occur on the same time step. This is also true for the formalism.

time step	Ping Agent	Pong Agent
1	Send Ping	Receive Ping
2	Receive Pong	Send Ping
3	Send Ping	Receive Ping
4	Receive Pong	Send Pong
...	...	,,,

Table 5.3: Expected output trace of the Ping Pong agents

5.1.3 Dynamic population environment

This testcase demonstrates the dynamic populations in the SARLforSIM tool. In this example we define a simple agent type called the SpawnerAgent. The purpose of this agent is to send an influence to create a new agent. This agent is defined as follows:

$$AgentBasedModel = \langle A_{abm}, AD_{abm}, E_{abm}, ED_{abm}, e, Z_{ij} \rangle$$

Where:

- $A_{abm} = \{id_0, id_1, ...\}$
- $AD_{abm} = \{SpawnerAgent_0, SpawnerAgent_1, ...\}$
- $E_{abm} = \{EnvId_0\}$
- $ED_{abm} = \{DynPopEnvironment_0\}$

$$SpawnerAgent = \langle I_{in}, I_{out}, S_{aw}, S_{cond}, S_{act}, P, D, A, ta \rangle$$

Where:

- $I_{in} = \emptyset$
- $I_{out} = \{\}$
- $S_{aw} = \emptyset$
- $S_{cond} = \emptyset$
- $S_{act} = \{spawnNewAgent\}$
- $P = \emptyset$
- $D = \emptyset$
- $A = \{(spawnNewAgent \rightarrow \{(SpawnInfluencem, EnvironmentID)\})\}$ Where EnvironmentID is the ID of the DynPopEnvironment in which the agent are situated.
- $ta = \{(spawnNewAgent \rightarrow 1)\}$

The Environment is then defined as follows:

$$DynPopEnvironment = \langle I_{in}, I_{out}, S_{env}, A, \delta_{env}, \delta_{react}, \delta_{conf}, \lambda_{env}, ta \rangle$$

Where:

- I_{in}
- $I_{out} = \emptyset$
- $S_{env} = \emptyset$
- $A = \{id_0\}$
- $\delta_{env} = \emptyset$
- $ta = +\infty$ for all inputs

While, δ_{env} and δ_{conf} are omitted for this simple example, δ_{react} is developed as follows: For each SpawnInfluence in the received influence buffer, a new id is added to A. This is the id of an agent that is not yet participating in the DynPopEnvironment.

In the formalism, we are not limited in the amount of agents we can define. Therefore, we can define an infinite set of agents that can be added to the environment during the simulation. However, in the SARLforSIM platform, we are limited to the available space of the device on which it is run, as well as the limited time developers have to implement the model. Therefore, agents are created and deleted at runtime.

For the test case, we want to see that our model indeed creates the new agents properly. When the simulation is run for 4 time units, we expect the total population to be 16. Indeed, if we run the example, we see an exponential growth of agents. In the output trace of the implementation, we printed out the string "New agent started", whenever a new agent is initialized. We see that the model outputs the expected amount of "New agent started" strings on the correct timesteps. The output trace of the model can be found in Appendix A.3.

5.1.4 Spatial environment - simple traffic case

For the last scenario, we want to demonstrate how a spatial environment can be implemented in our tool. To do this, we decided to recreate a model from the NetLogo Models Library. We chose the Traffic Basic model, which features a set of cars on a single road. At each time step, if a car notices that there is no car on the patch before him, the car will accelerate by increasing his speed with a user defined acceleration. The car cannot increase his speed above the maximum speed of 1. Whenever an car notices a car on the patch before him, he adjusts his speed such that it equals the speed of the car in front of him minus a user defined deceleration.

Because this model is more complicated than the previous ones, we will only give an overview of how this model can be implemented in our formalism. First of all, we need to define a car agent. This agent takes care of the behaviour of the car, which

Table 5.4: Expected output trace of the scenario with a single car

time	acceleration	speed	position
0	0.065	0.757871	0
1	0.065	0.822871	0.790371
2	0.065	0.887871	1.6457424
3	0.065	0.9528710086389911	2.566113
4	0	1	3.566113
5	0	1	4.566113
6	0	1	0.566113
...	...	...	...

is rather simple, if there is a car in front of the agent, it will send an influence that it wants to adjust its speed to match the other car. When the road is clear before him, he will send an influence that he wants to accelerate. Thus there are two types of input influences the agent can receive: RoadFree and RoadClear. The RoadFree influence has additional data of the speed of the car in front and the current speed of the agent. The agent also has a single type of output influence, DesiredNewSpeed, which contains the desired speed of the agent.

In this model, there is a single environment, the RoadEnvironment. The environment maintains all the facts and the global state of the situation in which the agents exist. In this model, the environments state consists of the positions of each of the agents and their actual speed and acceleration. Another part of the environment is the patches. The patches are used to determine the part of the environment that each agent can observe. Each agent only gets notified if there is an agent in the patch before him or not. The environment provides the agents with this information along with all observable parameters (the agent's speed and the speed of the car in front). Whenever the environment receives a DesiredNewSpeed influence from a car, the environment updates its facts about the car. During each transition of the environment, all positions, speeds, and accelerations of all cars are updated according to the time that is passed since the last transition.

In our first test, we wanted to see if the environment works correctly, when only a single agent is present. Figure 5.2 shows how the variables of the car change over time. The car's initial speed is 0.757871, its acceleration is 0.065, and its starting position is 0. The model that we adapted from NetLogo is updated at equidistant time-intervals. In Table 5.4, we discretized the graphs from Figure 5.2 into timesteps of length 1.

After running the simulation, we obtained an output trace similar to table. However, a small deviation occurs during the acceleration phase of the car. This is due to our discretization, in which we do not take into account that speed is increased gradually. The speed of the agent is considered constant between two timesteps. The position is updated after the speed of each agent. This results in a final position that is slightly

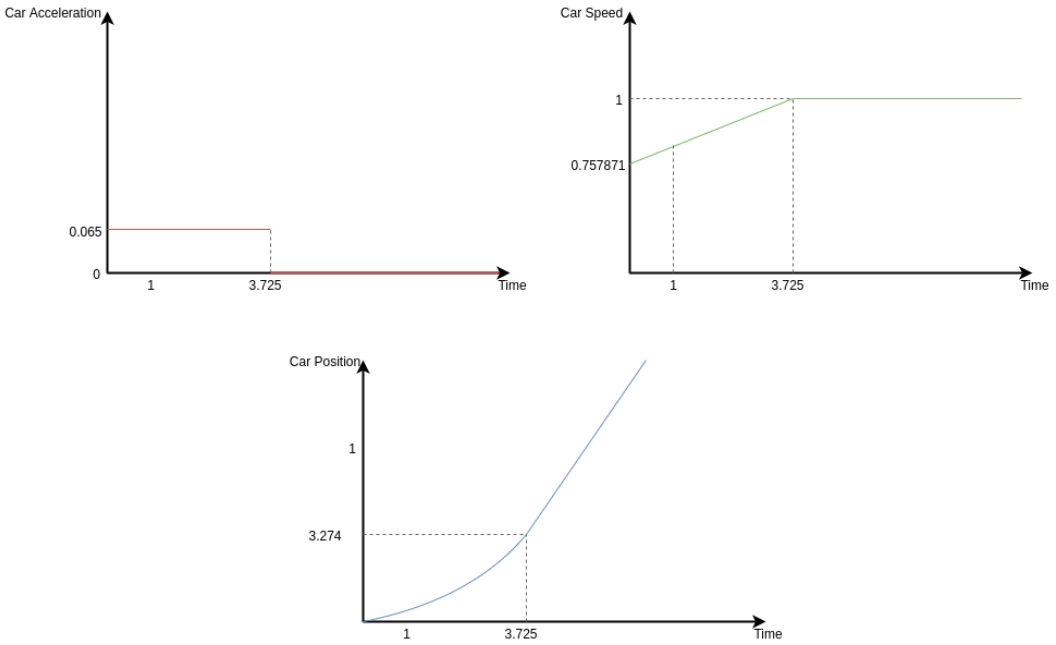


Figure 5.2: The graphs depict the rate of change of the car variables through time in the single car scenario.

higher than expected. The actual output of our model can be seen in Appendix A.4

For the second test, we wanted to check wether the model behaves correctly when two agents are placed in in subsequent patches, and the first agent is slower than the second one. In this scenario we have two car agents, one starts in patch 0 with speed 1, the other starts at patch 1 with speed 0.5. In Table 5.5 we list the output of our scenario. These values were checked and considered accurate, considering the errors made by our discretization.

5.2 Running example

To show how our tool compares to other agent-based modelling and simulation tools, we implemented the running example of Section 2.6 in different contemporary tools. The tools we used are: NetLogo, Mason, GAMA/GAML, and our own SARLforSIM tool. Our selection of tools is based on the selection made by Railsback et al. [70], as well as our own experience with agent-based modelling tools. We omitted tools that are very similar to the tools in the selection (such as relogo [65] and turtlekit [57]).

Table 5.5: Caption

	Car1		Car2	
timestep	speed	position	speed	position
1	0.475	0.475	0.565	1.565
2	0.54	1.015	0.63	2.195
3	0.605	1.62	0.695	2.89
4	0.67	2.29	0.76	3.65
5	0.735	3.025	0.825	4.475
6	0.8	3.825	0.89	0.365
7	0.865	4.69	0.955	1.32
8	0.93	0.62	1	2.32

5.2.1 NetLogo

The first tool in which we implemented the running example was NetLogo [79]. NetLogo has had a great impact on other agent-based modelling languages [65, 57], so we wanted to see how well this tool handles the running example.

Implementation

NetLogo allows modellers to create graphical user interfaces. Each user comes with a standard visualization of the model that shows each agents, the patches, and the links. Additionally other elements can be added to the interface to give more control to end-users and to feed back information. The representation of each entity is defined in the code of the model. Buttons allow end-users to invoke certain methods in the code. Sliders, switches, choosers, and inputs allow users to tweak the parameters of the simulation. Monitors, plots, and outputs provide information about the model and the simulation at runtime to the user.

The interface defined for the running example can be seen in Figure 5.3. The init-amount-goats slider sets the amount of goats that is initialized at the beginning of the simulation. Energy-max defines the maximum energy each goat can have. The higher the maximum value, the longer goats can survive without any food. The daily-energy-consumption defines how much energy a goat consumes during a single tick (in our model one tick equals one day). The smaller the loss of energy during the day, the less goats need to eat to refill their energy level. The three sliders food-coeff, mate-coeff, diffusion-coeff are used in the decision making process of the agent. For a high food-coeff, the agents will search for patches with a high amount of food during their migration. With a high mate-coeff, agents will prioritize patches where there is a possible mate. With a high diffusion-coeff, goats will avoid patches with a high population, aiming to increase diffusion and thus their food intake. The two plots on the right show the total amount of grass over all patches and the total

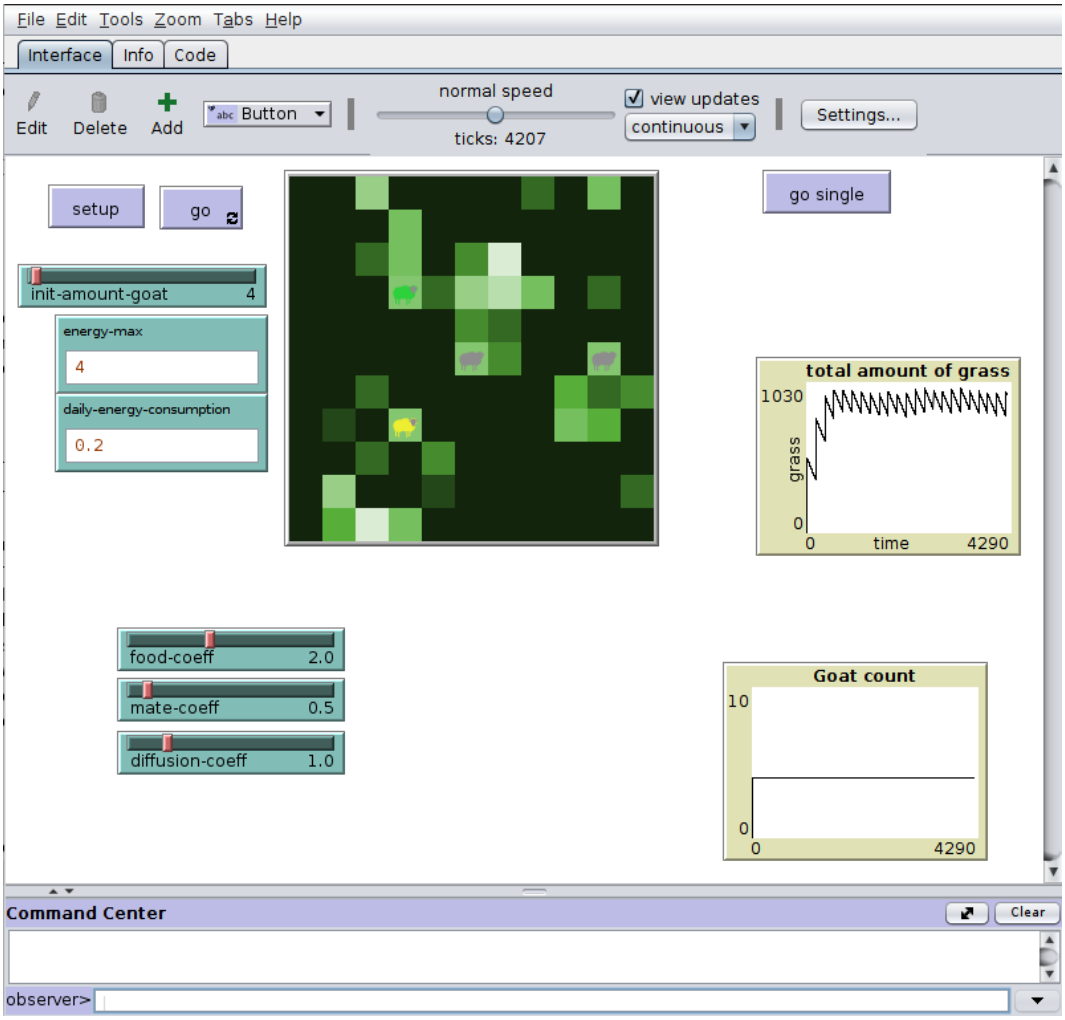


Figure 5.3: The running example in NetLogo

amount of goats that is present in the model. The central visualization shows the current state of the model. The intensity of the green color denotes the amount of grass that is present on the patch. Every goat is shown at the patch that it is currently at.

In the code tab of the model, you can define a series of methods that direct the simulation as well as define new breads and variables. We first defined a new breed called [goat goats]. In the goats-own statement we defined all fields each goat has:

- **energy:** The current energy of the goat.
- **ticks-till-lamb:** The amount of ticks before the goat will birth a new lamb (only used by female goats).
- **pregnant:** True if the goat is pregnant, false otherwise (only used by female

goats).

- **sex:** The sex of the goat.
- **lost-conflict:** True if the goat lost a battle during and has not migrated yet (only used by male goats).
- **age-counter:** The amount of ticks until the goat dies of old age.

We also defined that each patch has a field **grass-lvl**, which represents the amount of grass that is left in the patch. As for the global fields that are not defined by the input elements, we defined:

- **ticks-till-next-season:** The amount of ticks before the next season change.
- **current-season:** The current season, which is either summer or winter.
- **gestation:** The length of the gestation period of goat.

The method that can be called with the setup button sets up the simulation. It first resets the tick counter and clears the all turtles, patches, drawings, plots, etc. It also initializes all global variables. Ticks-till-next season is set to 190, current-season is set to "summer", and gestation is set to 150. Then it initializes all patches and sets the grass-lvl to a random number between 0 and 10. Finally it initializes the initial amount of goats at random locations. Pregnant and fight-lost are set to false. Each goat's sex is determined randomly and their age counter is set to a number between 5400 and 6480 days.

The go method controls what happens during the simulation of a single time step and can be executed repeatedly by toggling the go button. The first task of the go method is to update the energy levels and the age counters of the goats and checking if they survive this time step or not. The next step is to update the season. The ticks-till-next season is subtracted with 1 and if it is below zero, the timer is reset to 190 and the season is changed. If the season changes, the grass-lvl in each of the patches is updated. If the season is changed to spring or autumn, all patches increase their grass-lvl with a number between $[0; 4)$, in summer, $[0; 6)$.

The goats are updated each day. First all conflicts are resolved. Because, NetLogo does not have any built in mechanics for sending events or messages, this is done by inspecting each patch. If two or males are on the same patch. An agent is picked with the maximum energy. All other agents lose the fight and need to migrate from this timestep. The next step is that every agent will eat if there is still grass available, or migrate otherwise. During a migration, a goat picks a neighbouring patch that maximizes the function in Figure 5.4.

Finally, goats will try to procreate. If a female goat is not pregnant and she is on a spot with a male goat, she becomes pregnant and the ticks-till-lamb counter is set to the gestation period. If the goat is pregnant the ticks-till-lamb counter is adjusted. If the counter reaches zero, a new goat spawns at the same spot as the mother.

```
to-report heuristic [s]
  let mates any? goats-here with [s != sex]
  ifelse mates [
    report  (food-coeff * grass-lvl) + (mate-coeff) - (diffusion-coeff * count goats-here)
  ][
    report  (food-coeff * grass-lvl) - (diffusion-coeff * count goats-here)
  ]
end
```

Figure 5.4: The heuristic function for migration in NetLogo

Running the model

The model was created using NetLogo 6.0.4 and can be downloaded at this *link*[2]. The .netlogo file can be loaded into NetLogo and the model is ready to run.

5.2.2 GAMA/GAML

GAMA [38, 7] is very similar to NetLogo, it provides a complete development environment for creating and simulating agent-based models. Although there are a lot of similarities between the two tools, GAMA introduces concepts such as actions and reflexes. We were interested if this entails that semantics of agents are different from NetLogo.

Implementation

Instead of the observer agent in NetLogo, we find the Global agent in GAMA. The global agent maintains global variables and executes global reflexes or actions. Reflexes are executed each time step if the *when* condition evaluates to true. In our model, we did not define any global reflexes. As global variables we defined the same variables as in the NetLogo model. Similar to the reflex, the init statement groups a set of actions that are executed when the agent is created or, in case of the global agent, before the first time step is simulated. In init statement of the global agent, we create the initial number of goats.

Apart from the global agent, we created the goat species. A species defines a type of agent with its own fields, reflexes, and actions. The Goat species has the same fields or variables as the equivalent NetLogo turtle. Each variable can be given a parameter called update. This parameter defines how the variable is updated every time step. For example, the age counter is given $age\_counter - 1$ as update parameter. Every time step, $age\_counter$ is replaced by $age\_counter - 1$. For the goat species, we created two reflexes: The eat reflex, in which goats will either eat grass or migrate to another patch, and the procreate reflex, in which female goats try to bear offspring. In NetLogo, we have absolute control over the execution of the model. In GAMA, when two reflexes can be executed on the same time step, we cannot decide

[2]https://msdl.uantwerpen.be/git/tleys/RunningExamples/src/master/NetLogo

```
model testreflex1

global{
    init{
        create test number: 1;
    }
}
species test {
    reflex a {
        write "a";
    }
    reflex b {
        write "b";
    }
}
/* Insert your model definition here */

experiment prey_predator type: gui {
    output {
    display main_display {
    }
  }
}
```

(a) Test 1

```
model testreflex2

/* Insert your model definition here */

global{
    init{
        create test number: 1;
    }
}
species test {
    bool t <- false;
    reflex a when: t {
        write "a";
        t <- false;
    }
    reflex b when: !t {
        write "b";
        t <- true;
    }
}
/* Insert your model definition here */

experiment prey_predator type: gui {
    output {
    display main_display {
    }
  }
}
```

(b) Test 2

Figure 5.5: Test models to check the execution order of reflex statements

the order in which they are executed. After running the example in Figure 5.5a, we concluded that reflexes are executed in the order they are defined. The second reflex also sees the change of state after the first reflex has been completed. For example, the test in Figure 5.5b outputs "b" in the first timestep, but afterwards always "ab".

Contrary to NetLogo, GAMA does not have a default grid environment. To add this to our model we need to define a grid species. This species defines the variables, reflexes, and actions of each cell in the grid. Each cell maintains its current grass level as well as the current season. For the patches, we defined two reflexes, the resolve_conflicts reflex, which resolves all fights between bucks on the same patch, and the change_season reflex, which updates the grass levels and season every 190. Since GAMA only allows discrete time models, we needed a counter to determine when the next season change happens.

The visualization of the model is defined in the experiment statement. Here we can define input parameters for the model, and output visualizations and charts. For simplicity, we omitted the input parameters. For the output we created a visualization that shows every patch as a green square and each goat as a blue dot, as seen in Figure 5.6. The intensity of the green indicates the current grass-level of the cell.

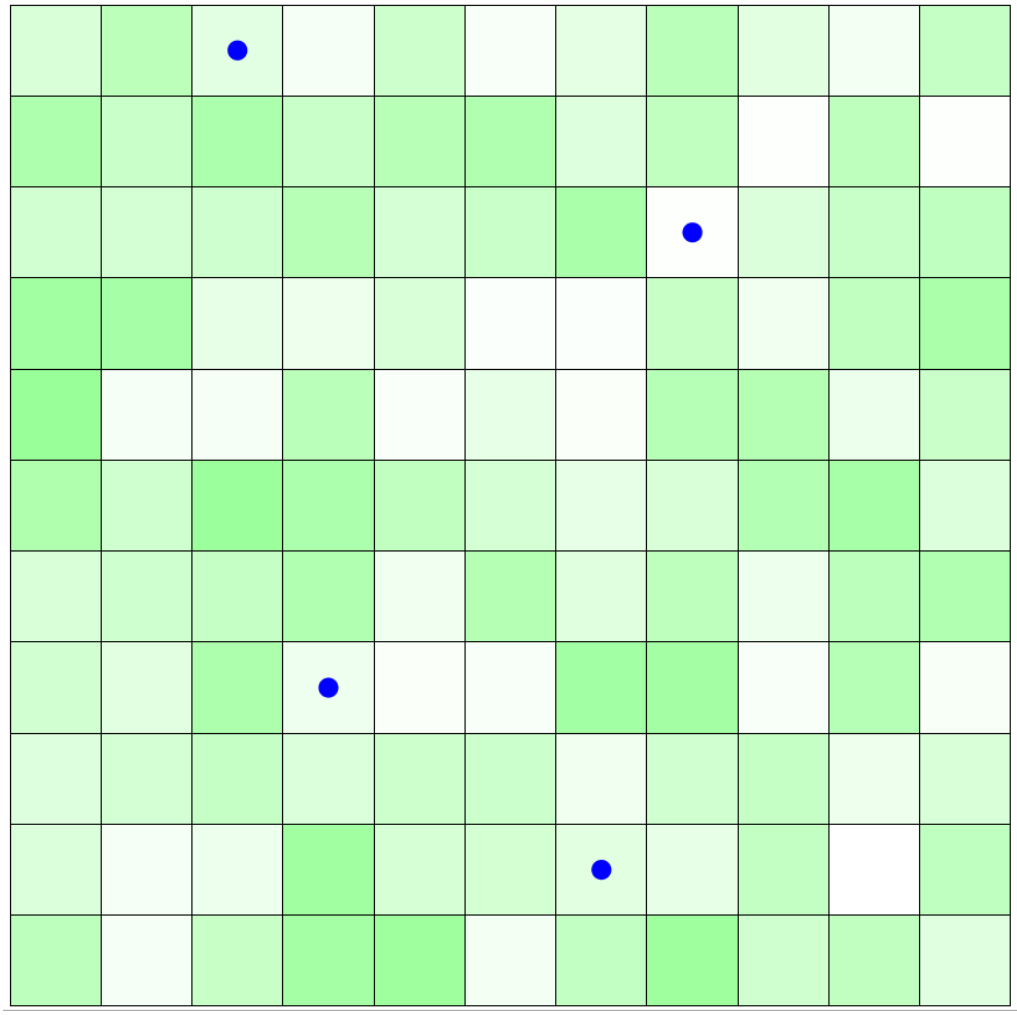


Figure 5.6: The visualization of the goat model in GAMA

Running the model

The model was created using GAMA 1.8.0. To run the model, download the .gaml from this *link*[3], then load it into GAMA.

5.2.3 MASON

MASON [49] provides an extensive library for developing agent-based models. Instead of providing a specific language for developing agent based models, MASON provides an internal domain specific language in Java. We were interested whether MASON approaches agents and environments differently than Logo based platforms.

[3]https://msdl.uantwerpen.be/git/tleys/RunningExamples/src/master/GAMA

Implementation

The central entity in a MASON model is the SimState. The SimState provides a discrete event scheduler, a random number generator, and functionality to execute the simulation loop. To create a model, the SimState can be extended. As seen in Figure 5.8, we extended the SimState in the GoatSimState class. In this class we define global parameters of the model, similar to the globals in NetLogo. These globals comprise:

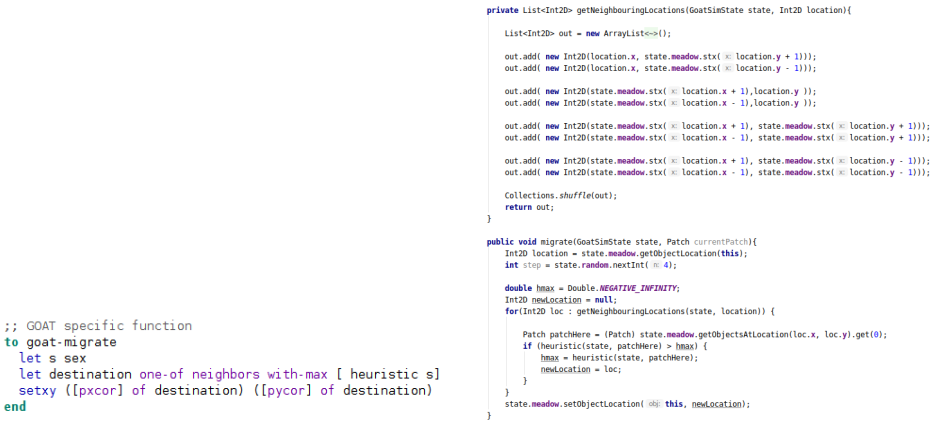
- **initAmountSheep:** the initial amount of sheep in the model
- **maxEnergy:** the maximum energy of a goat
- **dailyEnergyConsumption:** The energy each of the goats consume every time unit
- **meadow:** The sparse grid that represents the environment

Contrary to NetLogo, MASON does not have a default environment for the models. However, multiple built-in types can be used to represent the environment. We chose to use a sparse grid, because it allows multiple objects to be placed in the same cell.

The SparseGrid2D does not allow to assign variables to cells, thus we created the Patch type which takes over the role of the patches in the NetLogo model. Each patch has two fields: its grass-lvl and the current season. A patch implements the steppable interface, which allows us to schedule the patches at an interval of 190 timesteps to update their season and grass-lvl.

Apart from the patches, we also defined the Goat type to represent the goats in the model. The Goat type has the same fields as the Goats in the NetLogo model. Agents also implement the steppable interface and are scheduled to execute their step at each timestep. Goats, however, schedule themselves on each time-step. If the goats are scheduled at the start of their lifetime, an error occurs when the goat dies unexpectedly due to lack of energy, since there is no way of deleting a scheduled event. During a time step, the agents perform the same steps as in the NetLogo model. Firstly, the energy level and age counter of each goat is updated. Then, all conflicts between bucks is resolved. Conflicts are resolved by inspecting each cell in the grid. If multiple bucks are present, the buck with the most energy is chosen, all other goats set their lost_fight variable to true. Goats eat or migrate accordingly. Finally, the reproducion phase is performed.

Although the workflow of MASON is similar to NetLogo, MASON does not feature a lot of specific methods to facilitate the implementation of the model. This is very prominent in the implementation of goat migration. In Figures 5.7a and 5.7b, we show the implementation in both NetLogo and MASON. In MASON we need to calculate all neighbouring patches ourselves as well as determine the patch that maximizes the heuristic function. In NetLogo a single built-in method can be used to replace each step and they can be combined into a single line of code.



```
;; GOAT specific function
to goat-migrate
  let s sex
  let destination one-of neighbors with-max [ heuristic s]
  setxy ([pxcor] of destination) ([pycor] of destination)
end
```

```
private List<Int2D> getNeighbouringLocations(GoatSimState state, Int2D location){

    List<Int2D> out = new ArrayList<>();

    out.add( new Int2D(location.x, state.meadow.stx( o: location.y + 1)));
    out.add( new Int2D(location.x, state.meadow.stx( o: location.y - 1)));

    out.add( new Int2D(state.meadow.stx( o: location.x + 1),location.y ));
    out.add( new Int2D(state.meadow.stx( o: location.x - 1),location.y ));

    out.add( new Int2D(state.meadow.stx( o: location.x + 1), state.meadow.stx( o: location.y + 1)));
    out.add( new Int2D(state.meadow.stx( o: location.x - 1), state.meadow.stx( o: location.y + 1)));

    out.add( new Int2D(state.meadow.stx( o: location.x + 1), state.meadow.stx( o: location.y - 1)));
    out.add( new Int2D(state.meadow.stx( o: location.x - 1), state.meadow.stx( o: location.y - 1)));

    Collections.shuffle(out);
    return out;
}

public void migrate(GoatSimState state, Patch currentPatch){
    Int2D location = state.meadow.getObjectLocation(this);
    int step = state.random.nextInt( n: 4);

    double hmax = Double.NEGATIVE_INFINITY;
    Int2D newLocation = null;
    for(Int2D loc : getNeighbouringLocations(state, location)) {

        Patch patchHere = (Patch) state.meadow.getObjectsAtLocation(loc.x, loc.y).get(0);
        if (heuristic(state, patchHere) > hmax) {
            hmax = heuristic(state, patchHere);
            newLocation = loc;
        }
    }
    state.meadow.setObjectLocation( obj: this, newLocation);
}
```

(a) NetLogo (b) MASON

Figure 5.7: The implementation of the migrate function

MASON allows developers to create a specific GUI for their models. This is done by extending the GUIstate. We created the GOATSimStateGui, which extends the GUIState and initializes it with a new GOATSimState. To create a visualization, a portrayal needs to be set for the grid of the model state. Within the field portrayal, we can set the portrayals of the patches as well as the agents. Similarly to the other models, patches are green squares. The intensity of the square indicates the amount of grass still present on the patch. Goats are represented by a grey dot on the patch.

Running the model

We implemented the model using IntelliJ IDEA 2019.3 community edition, JAVA 11.0.5, and Mason 20. The project can be downloaded on this *link*[4].

5.2.4 SARLforSIM

Finally, we implemented the running example in our own tool. SARLforSIM differentiates itself from the other tools in that its semantics are defined precisely.

Implementation

To represent goats, we created the GoatAgentModel, which extends the AgentModel. To enable the agent to reason about its decision, we needed it to have a local representation of its environment. His local representation of its environment comprises its current location, the grass level on the current location and the neighbours, as well as the amount of male goats on those patches. The range of the neighbours are

[4]https://msdl.uantwerpen.be/git/tleys/RunningExamples/src/master/MASON

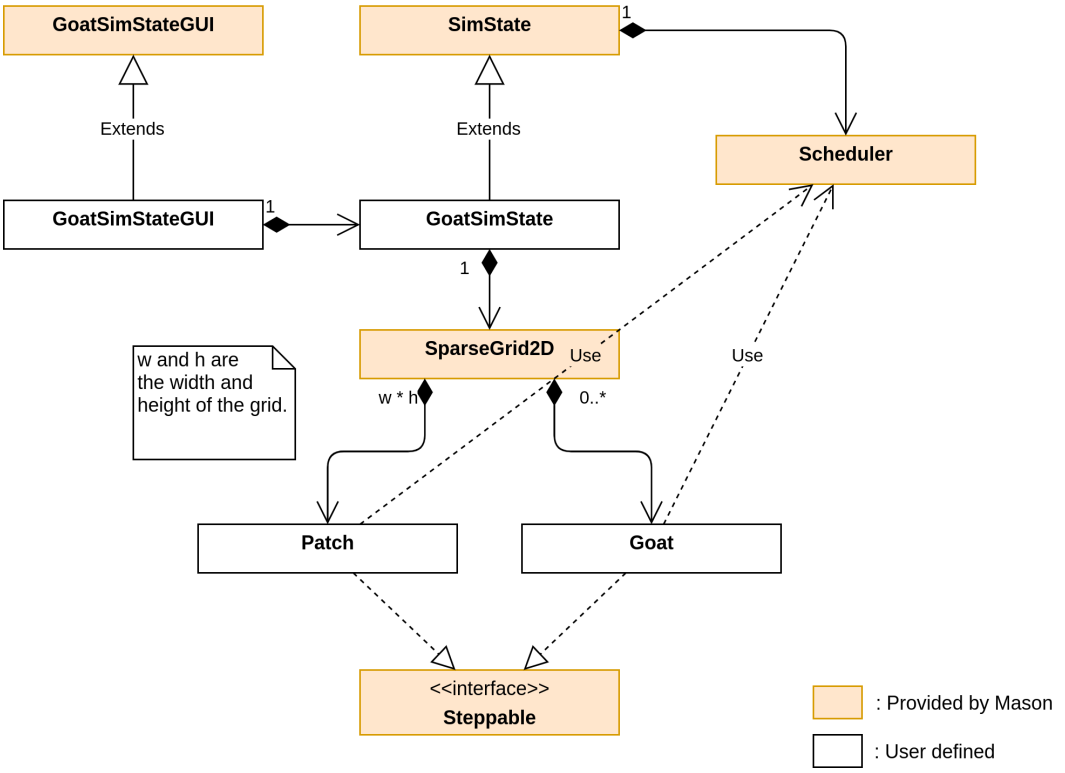


Figure 5.8: The design of the mason model

defined by the environment, in our case these are the eight direct neighbours of the current position. Furthermore, each goat keeps track of its own energy level as well as the parameters in the goat parameters of the NetLogo model.

To implement the perception step, we have two options. The first option is that the environment sends percept events periodically to the agents. The second option is that agents send a perceive request to the environment, which in turn replies to the request. Although the first option is easier, we chose the second option because it resembles the real world more truthfully. So the first step of the Goat agent at a certain time step is to send a perceive event to the environment. Since the agent does not know whether he will get a response from the environment, it sets its action state to Perceive, and its time advance to 1. This way the agent schedules the perceive action for the next day. The agent can receive three types of influences from its environment. The first one is the perceive response. The perceive response updates all variables in the awareness state of the goat. When the agent perceived such a response, he has the information to make the decision for his next action. The second influence is the EnergyConsumed influence, this influence tells the agent the amount of grass the agent consumed and how much energy he recuperates. When this influence is received the variable *update* is set to true. This tells the decide function that the last perceived influence does not contain new information for making an action decision and that the conditional state and action state can

remain the same. The last influence is the FightLost influence, which tells the agent that he lost the fight. Because he can only migrate during the next day, this is also considered an update.

The decision function is implemented as follows: When the update is set to true, the state is set to perceive, since no decision can be made on the new information of these events. If no perception has taken place during this simulation step, the agent has just send an influence to the environment. Again, since the agent is not certain an answer will come, he sets his next action to Perceive. In all other cases the agent will make a decision to act on its environment. When the amount of grass is more than 0, he will set his action state to EAT. When it is 0 or when the goat has lost his last fight, he will set is action state to migrate. If a buck notices other bucks, it will set its action state to fight. If his energy is below 0, he will set his action to die regardless of any perceptions. If a goat is female and a buck is present in the patch, the doe will set its pregnant field to true and its time until birth to 190. Whenever a perceive request is sent, we update the time until birth, since 1 day has passed since the last perceive request. If the timer reaches zero, the goat will send, additionally to its action, a Birth influence to the environment.

After the decision function, the agent specifies how long it will remain in its current action state. Whenever the agent is in the perception state, meaning that he will perform a perception request action next, he will wait for 1 time unit (day). In all other cases, he will send the action immediately.

In the action function, the appropriate influences are sent to the environment. When the current action is set to Perceive, a PerceiveRequest is sent. When the next action is set to Eat, an Eat influence is sent with the desired amount of energy the goat wants to consume. When the goat wants to migrate he sends a Migrate influence containing the desired destination. If he wants to fight, a fight influence is sent with the current energy level of the agent. If the energy levels of the agent drop below zero, the agent sends a Die influence to the environment.

Our platform does not feature a ready to use spatial environment like NetLogo, however, we provided the AbstractToroidalGrid2D, which provides users with a blueprint of an environment where objects can be placed in a sparse toroidal grid. We extended this class into the MeadowEnvironment class, which we use as environment for the goat agents. According to the width and height specified in the constructor, the environment places a Patch object on each cell of the toroidal grid, similar to the MASON model. A Patch object contains and maintains the grass level in that cell of the grid. If a GoatAgent is added to the model, its ID is placed in the grid to remember the agent's position. Also, the environment maintains a map of goat IDs to their sex. The final field that makes up the global state of the model is the current season.

The MeadowEnvironment has two purposes, the first is to update the grass levels in the patches and change the season. This happens regardless of agent interaction and is modelled in the environment transition function. The other purpose is to

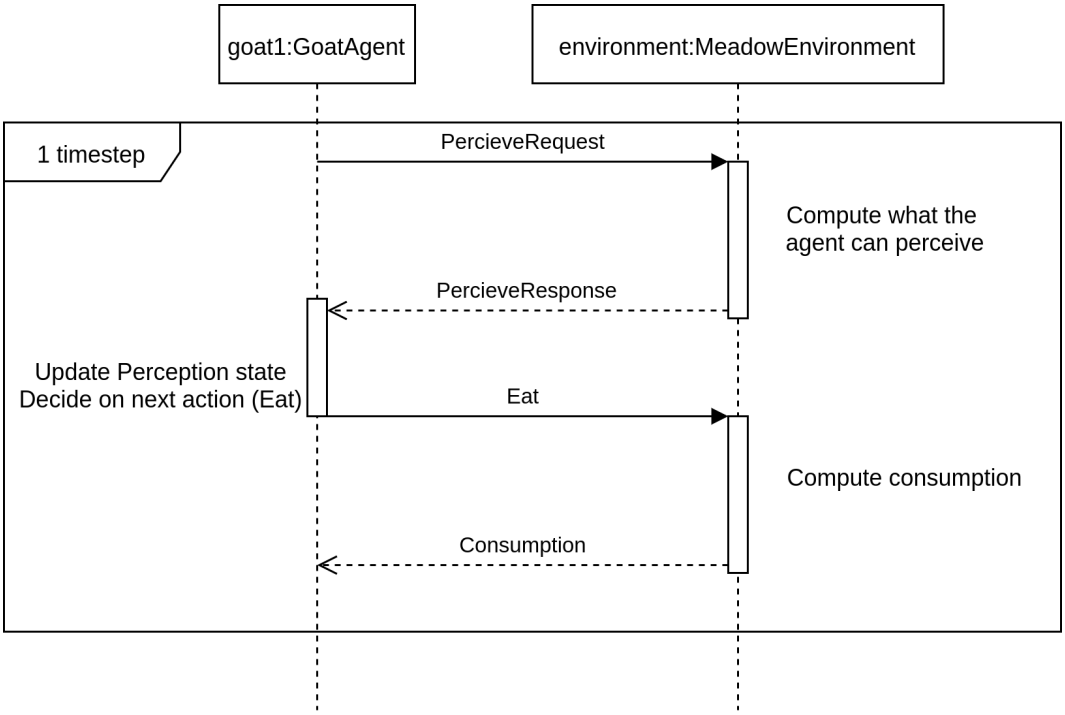


Figure 5.9: A sequence diagram of a timestep in which a single goat agent performs the eat action

react to agent influences and send back necessary information. This is implemented in the reaction transition function. Whenever a Perceive request is received, the environment will respond immediately by sending a perceive response back. For each eat influence that is received by the environment the environment calculates how much is still available and sends back how much the agent has consumed. Because we have a non-deterministic environment, goats do not know up front if there will be grass available when multiple goats try to eat from the same patch. The environment updates all agents that send an eat influence with the amount of energy they recuperate from eating. The environment also collects all fight influences sent that time step. Afterwards it calculates all losers of the fight and sends them a FightLost influence. For each Migrate event, the position of the agent is updated. No answer is sent to the agent, since he needs to perceive its environment first to know where he is. When a Birth event is sent, the environment creates a new goat agent and places it on the position of its mother. For each Die influence, the sender of the influence is removed from the environment as well as the model context.

Running the model

To run the model, download the SARLforSIM project on this *link*[5]. Then open the project in SARL 0.10 and make sure that Java version 1.8 is used. In the folder DemonstrativeExample, run the Runner class as SARL application.

5.2.5 Discussion of the running examples

In Section 2.6, we presented a set of requirements that the running example needs to fulfill. The requirements are based on essential or important aspects of agent based modelling. For example, in agent-based modelling, single entities in the model should be able to make complex deliberations for their next action based on their internal state and representation of the world. In this section, we will discuss in more detail how each of the different tools handles these aspects.

Reactive behaviour

Reactive behaviour allows agents to react to inputs or events of its environment. To support reactive behaviour, we would expect that an agent-based modelling tool features a dedicated way to notify agents of events that happened within their perception scope. However, NetLogo, GAMA, and MASON do not support any mechanism of the sort. Of course, due to the versatility of all these tools, it is implementable, but it is not a built-in feature. SARLforSIM agents can react to influences by reacting to the Perceive event, send by the simulator.

Proactive behaviour

Proactive behaviour allows agents to create opportunities for themselves, even when no events happen in the environment. To achieve proactive behaviour, agents should be able to schedule state changes and actions even if no events occur. GAMA only features discrete time, and although NetLogo allows tick sizes of variable length, discrete time modelling is the preferred method. In these tools all agents are given the option to perform a state change at each time step. In GAMA this is done by setting the *when* parameter of a reflex. NetLogo allows developers to specialize an agent set that is passed on to the ask statement.

MASON allows more options due to its discrete event scheduler. In MASON we can schedule an event to happen periodically or at a single point in time. The interval and start time are arbitrary and can be any floating point number. Because steppable objects receive the full state of the model, and thus the scheduler, agents can add new events at run time.

[5]https://msdl.uantwerpen.be/git/tleys/SARLforSIM

SARLforSIM also features a discrete event scheduler, however it is not directly controllable by the entities. Instead, agents schedule their next action with the time advance function. Herein the agent returns the time that it will remain in its current state before sending an action or changing its state.

Complex deliberation

A signigicant feature of agents is their ability to make complex deliberations about the action they will perform. NetLogo and MASON do not feature any built-in constructs or tools to facilitate this process. GAMA, on the other hand, features multiple control architectures which allows users to use the architecture that is most appropriate for the behaviour that needs to be implemented.

In SARLforSIM, we divided the process of making a decision in three steps, the perception step, the decision step, and the action step. In the perception step, the agent updates his local representation of its environment. In the decision step, the agent decides his next action based on everything he knows, and in the action phase he sends an influence to the environment. Though this is not as extensive and versatile as GAMA's control architectures, it gives developers a foundation for the decision making process.

Agent interaction

An essential feature of systems that feature multiple agents is agent interaction. In multi-agent systems, agents need to work together to achieve their goals. In agent-based modelling, the purpose is to model a system with a population of agents. Since multiple agents are involved, we expect that their interactions are represented in the models.

NetLogo and GAMA have the same approach to handling interaction, which is the *ask* statement. The ask statement allows the agent which executes it to execute a block of code in another agent or a set of agents. This method of interaction is identical to method invocation on an object in object oriented programming.

In MASON, the step method of the Steppable interface is passed the current state of the model as parameter. Steppable objects can interact with each other by invoking methods are altering public variables.

SARLforSIM implements the influence reaction model. This means that agents interact with their surroundings by sending influences. Then the reaction is computed based on all influences sent that time step. Agents in SARLforSIM cannot interact with each other directly, but need to send an influence to an environment in which they are situated that they want to contact a certain agent. The environment then decides whether the message reaches the other agents and forwards the message accordingly.

Dynamic population

A characteristic feature of agent-based modelling is that the population of the model can change during the simulation. Agents can die, as well as spawn other agents.

In NetLogo, two methods are defined to create new agents. The *create-turtles* method is used to create a set of turtles from the Observer agent. When a turtle wants to create new turtles, the *hatch* method is used, which initializes other agents with the same parameters as the spawner unless specified otherwise.

In GAMA, a similar method as the create-turtles method is available, the *create* method. This method can be called from any agent.

MASON does not have any built-in methods to create new agents. However, new steppable objects can be added to the model and scheduled at run time.

In SARLforSIM, we provide a the AbstractDynPopEnvironment. This environment can add new agents to or remove agents from the simulation. Agents can send an influence to spawn a new agent or the environment can add agents to the simulation dynamically. There is no control over which agent can spawn which other agent.

Dynamic environment

A dynamic environment is an environment that can change its state without any agent interference. In NetLogo, dynamic environments cannot be implemented. Everything is directed by an agent. We can however implement the model, such that the patches change their state without turtle interference, which resembles a dynamic environment. Similarly in GAMA, we can implement reflexes in the agents that represent the environment.

In MASON, fields do not implement the steppable interface. This means that fields cannot be scheduled. However, because MASON is a Java library, we can utilize Java's full capabilities and extend an existing field which also implements the Steppable interface. In the running example, however, we used the same technique as NetLogo where we implemented the patches as steppable objects.

In SARLforSIM, environments can specify when they will change their state if no external influence is received in the time advance function. If they reach that time without interruption, they execute the environment transition function.

Non-deterministic environment

In a non-deterministic environment, agents do not know the outcome of their actions. In NetLogo, GAMA, and MASON all agents have access to the complete state of the model. This means that non-deterministic environments are impossible to represent in these tools.

In SARLforSIM, agents can only perceive their environment if the environment sends influences to the agents containing updates on the state of the environment. If an environment does not update an agent on the effects of the influence the agent has sent, the agent does not know the effects of his actions.

Partial view

Agents are often associate with space. They are situated on a certain location and can move through their environment. As in real life, agents therefore can only perceive a part of the environment in close proximity to the agent itself. Because all agents in NetLogo, GAMA, and MASON have full access to the complete state of the environment, a partial view of the environment cannot be implemented.

In SARLforSIM, agents perceive their environment by receiving influences. It is the environment that decides, or computes what the agent is able to perceive. In the running example, the PerceiveResponse only has information about the neighbouring patches of the agent.

5.3 Reproducibility analysis

The final evaluation we performed was a reproducibility analysis of the running example in the four tools. Herein, we will investigate whether the simulation remains stable under different parameters.

5.3.1 Stability analysis of a system with a fixed set of goats

In this analysis, we investigate how the system behaves with a fixed set of goat agents at the start of the simulation. We set the sex of each goat to female, such that no new agents can be spawned during the simulation, and removed the age counter for the simulation, such that agents can only die because they cannot find food. This gives us an idea of the amount of agents that can exist in the system, such that the overall resource does not get depleted over time. Each simulation was run for 5400 time steps (days) or a little over 14 years in virtual time. The parameters of the model are as follows:

- maximum energy: 4
- start energy: 4
- food coefficient : 1
- mate coefficient : 0
- diffusion coefficient: 0

The diffusion coefficient was set to 0 to keep the simulation simple. Agents will always prefer patches with the most food.

The initial amount of goats was increased gradually and for each initial amount of goats, the simulation was run 5 times with different seeds. The used seeds in each platform where $[1, 2, 3, 4, 5]$. After running the experiments, we inspected several properties of the system. These properties include: the amount of grass in the system, the size of the population, the sum of the energy of all agents, the amount of grass that is added to the system, and the total consumption.

Analysis of the system in a stable configuration

We noticed that in each tool, the state remains stable when there are 8 agents present in the model. Generally speaking, we saw no agent die of starvation, except in one simulation of GAMA, as seen in Figure 5.11.c. After investigating, we saw that in this particular simulation, the agents had eaten a corner, which is seen in Figure 5.10. GAMA also does not feature a toroidal grid such as NetLogo's default environment. This makes corners more dangerous and explains the agent's death.

If we inspect the averages of the total amount of grass that is produced during the simulation, and the total amount of grass that is consumed, we see that these are very similar to each other in each tool, which indicates that the system is in a stable state. Note that the added grass represent the amount of grass that is added after the initialization of the patches with their initial amount of grass. The amount of food consumed is replenished throughout the seasons. Notice that GAMA's consumption is lower than the others. This is the effect from the single agents that dies during one of the runs. The amount of grass that is replenished also corresponds to the expected value. There are approximately $5400/95 \approx 56$ season changes, so each season occurs approximately 14 times. The expected increase of grass on a single patch is 1.5 in spring and in fall, 2.5 in summer, and 0 in winter. The expected amount of grass that is added is given by:

$$(14 * 1.5 * 121) + (14 * 0 * 121) + (14 * 1.5 * 121) + (14 * 2.5 * 121) = 9317$$

All values are slightly lower than the expected value. This is because not every patch needs to be added the full amount each time. If we look to the example in Figure 5.10, we see on the right side a set of patches that are still pretty full while the agents have a lot of patches around them that are pretty full, such that they can survive until the next season without needing the top right patches.

In Figure 5.11, we show the averages of the agent population and the total grass amount through time. A difference we see here is that in NetLogo, the amount of grass of the system in the initial state is approximately 50 higher than in the other tools. When we inspected this in more detail, we saw that there are 121 patches and that they were correctly initialized with a random number in the interval $[0 : 10]$. Our explanation is that NetLogo's random number generator has a slightly higher

Table 5.6: A comparison of the total grass produced and total consumption

Tool	Total grass produced	Total grass consumed
NetLogo	8561.56	8641.56
GAMA	8548.16	8527.56
MASON	8623.6	8643.12
SARLforSIM	8578.04	8634.4

expectancy than the other tools, when random numbers are generated from this range.

Analysis of the system in an overpopulated configuration

For all tools, we see that when 9 goats are present from the start, the total consumption exceeds the amount of grass that is generated each season. This results in a slow, but steady decay of the total amount of grass in the system, until some agents die. In Figure 5.13, we plotted the average population and amount of grass in each tool.

Here we see an observable difference between the different tools. Here we provide our arguments for the difference in output.

As we look at the difference between NetLogo and GAMA, we see that on average, agents die at a sooner time step in GAMA. In GAMA, most agents die around time step 2000, while in NetLogo, the agents start to die at time step 3000. The reason for this is that the grid in GAMA is not toroidal. When agents go to the corner of the grid, they do this by choosing a path with the highest amount of grass. However, when they reach the corner, they cannot continue and need to go back. This time they will go to the patches they left alone when reaching the corner. These are the patches with less grass on. This pattern has two effects. Firstly, on their way back, agents have a greater risk of dying. Secondly, agents eat more patches with less grass on. This leaves more patches that are full during the next season change. Patches that are full are not replenished, thus the amount of grass that is added is far less than expected. This results in the quick death of the agents.

Both NetLogo and MASON produce very similar output. However in a previous version of the MASON model, we saw that agents die at a later point in time, and the dip in the function comes at a later point in time, as seen in Figure 5.12. The cause of this difference was that the list of neighbours was shuffled before the patch is selected that maximizes the heuristic function. This was done to mimic the NetLogo function *one-of*, which is used to determine the new destination if more than one neighbouring patch produce an equal maximum value. One-of, according to the NetLogo documentation [1], returns a random patch from the provided set. However, when inspecting the results of our analysis, we interpret that there is some

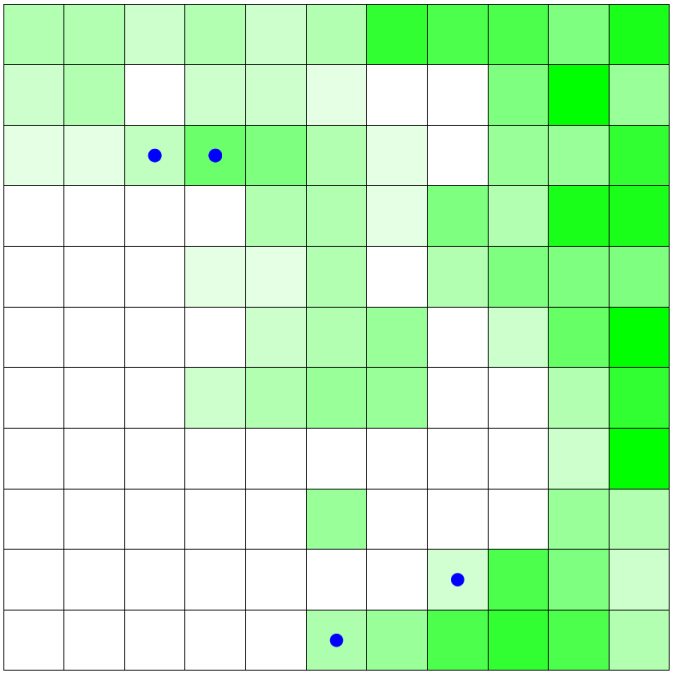
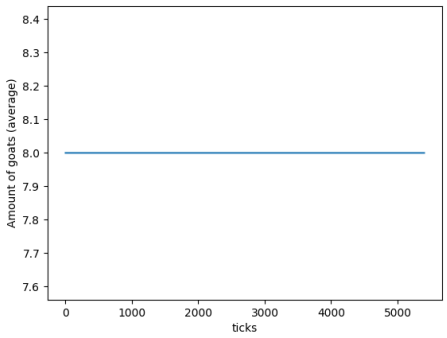


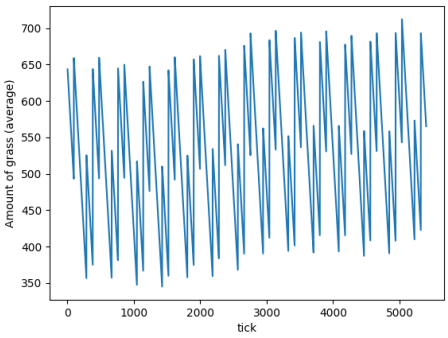
Figure 5.10: The situation in which one agent dies in the GAMA model with 8 initial goats. When multiple agents are on the same spot, only a single blue dot is shown. Therefore, only 4 dots are shown.

kind of bias in the randomness of the function, since we only get reproducible results in MASON if we eliminate that randomness factor. This emphasizes the need for precise semantics in ABMS tools. Due to the high amount of individual entities and variability that is often seen in agent-based models, small changes can result in significantly different output as demonstrated by this small case study and this related study performed by Donkin et al. [27]. If a tool has precise semantics, it would be much easier to pinpoint certain mistakes or variation points.

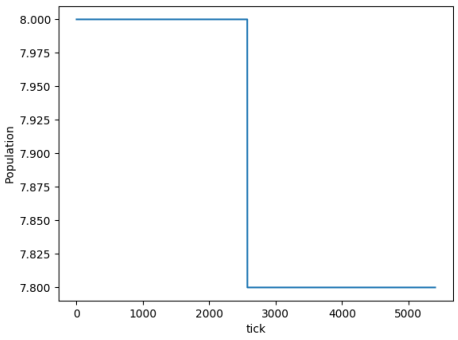
In our tool, we see a difference between NetLogo and MASON. Although the majority of agents die around time step 3000, we see that in SARLforSIM agents start to die much sooner then expected. We argue that this is due to our translation of the model into the influence reaction model. Here, all agents will first perceive their environment, then based on their perception and then they send the eat influence simultaneously. This means that in the case that there is not enough food for everyone, some agents decide to stay and eat, but do not get the chance to actually eat. This starvation can lead to the death of some agents on an earlier point in time. When a small set of agents dies relatively fast, the system can restable itself more easily. This can be seen in the average population of SARLforSIM, which is higher than in NetLogo or MASON, and also in the plot of the amount of grass, where the sawtooths after timestep 3000 are bigger in amplitude than in the other tools.



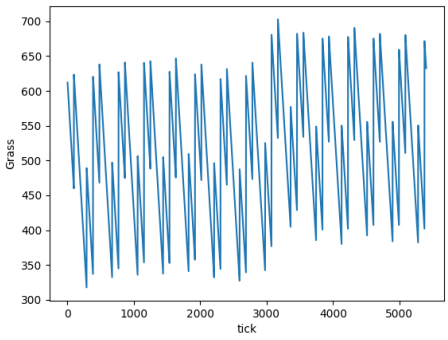
(a) The average population in NetLogo



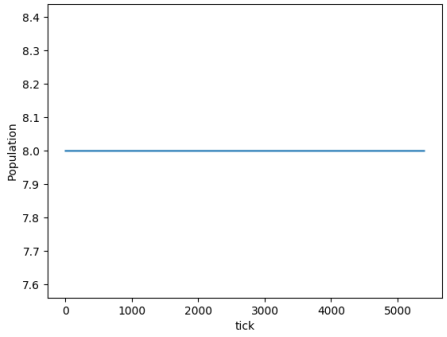
(b) The average amount of grass in NetLogo



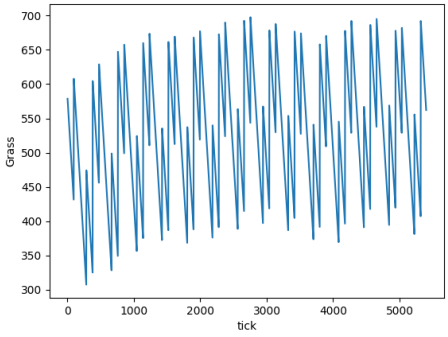
(c) The average population in GAMA



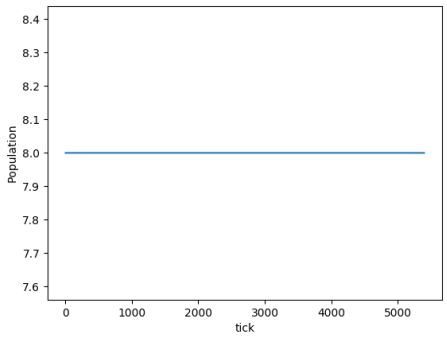
(d) The average amount of grass in GAMA



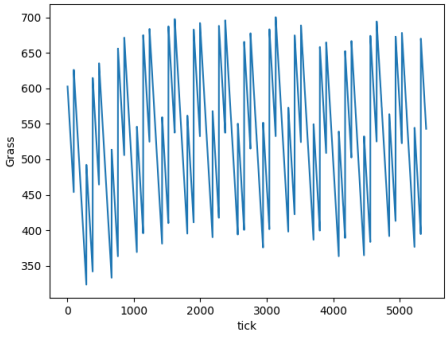
(e) The average population in MASON



(f) The average amount of grass in MASON

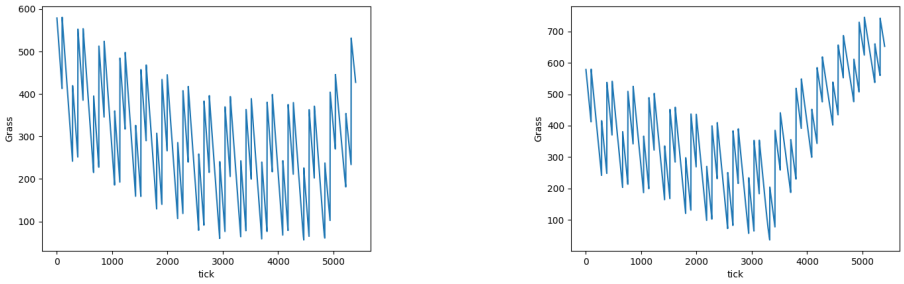


(g) The average population in SARLforSIM



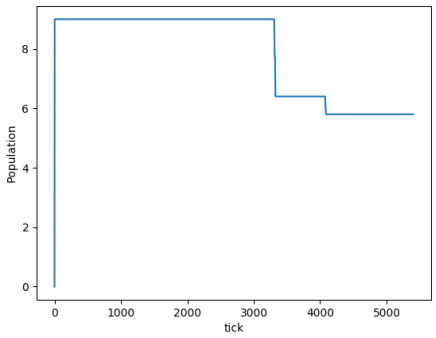
(h) The average amount of grass in SARLforSIM

Figure 5.11: The averages of the goat population and total grass amount with 8 initial agents

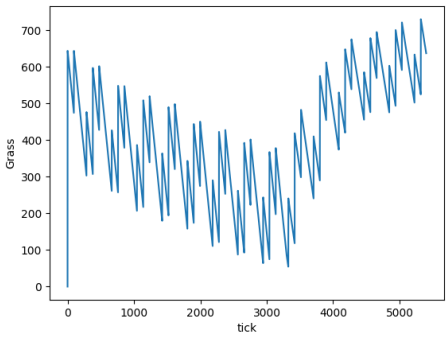


(a) With randomizing the neighbours (b) Withouth randomazing the neighbours

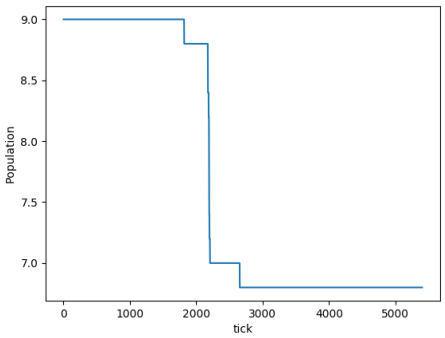
Figure 5.12: The plot of the average amount of grass in MASON



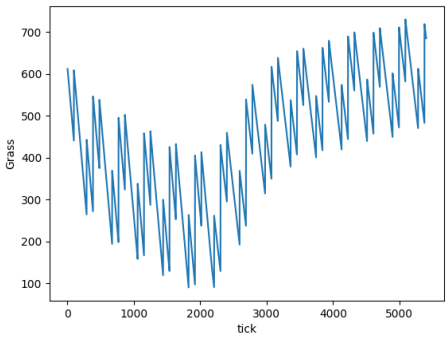
(a) The average population in NetLogo



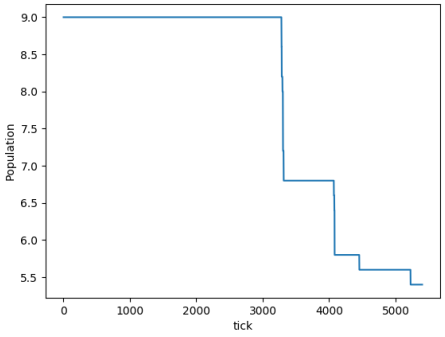
(b) The average amount of grass in NetLogo



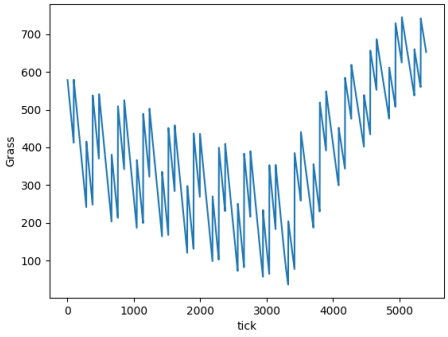
(c) The average population in GAMA



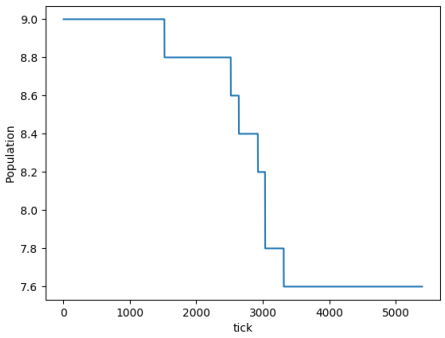
(d) The average amount of grass in GAMA



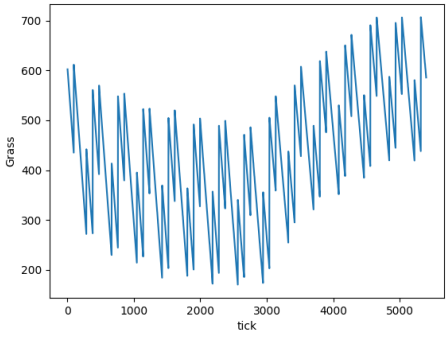
(e) The average population in MASON



(f) The average amount of grass in MASON



(g) The average population in SARLforSIM



(h) The average amount of grass in SARLforSIM

Figure 5.13: The averages of the goat population and total grass amount with 9 initial agents

CHAPTER 6

Related Work

The field of agent-based modelling is very broad. Here we present a coherent overview of the most prominent literature on which we have based ourselves on during this thesis.

6.1 Surveys on agent-based modelling and simulation

To get a better understanding of the agent-based modelling and simulation, we studied a set of literature reviews as well as tutorials on agent-based modelling. We list them here, since they are great for entry points for information about the paradigm.

In [51], Charles Macal provides an overview all the domains that use agent-based modelling and discusses how this approach is used in the field. Macal also provides four definitions for an agent-based model, each more extensive than the previous one, to categorize the various agent-based models that can be found in literature.

In [52], Charles Macal and Michael North discuss the concepts of agent-based modelling in more detail. They discus in detail the elements of an agent-based model, as well as the properties and tasks of an agent and the agent's environment.

Fabien Michel [58] discusses in great detail the need for modelling and simulation of multi-agent systems. In section 1.3 the history of agent-based modelling is explained in great detail. The focus in this paper is the applicability of agent-based modelling for investigating multi-agent systems.

Where the previous papers define agent-based models by their operational semantics, Eric Bonabeau [14] provides a categorization based on the type of system that is represented by the model. This paper shows clearly in which situations agent-based models are applicable or recommended.

6.2 Other tools and platforms

Agent-based modelling is a well established modelling paradigm, with an extensive collection of tools that enable modellers to create and simulate agent-based models. Here we present other agent-based modelling tools which we encountered during our study. For a list of almost the entire spectrum of agent-based modelling and simulation tools, we recommend reading the survey in [3]. Another interesting survey is presented by Railsback et al. [70]. Herein, they compare a selection of agent-based modelling tools by implementing an increasingly extensive model in each of them.

A very well known tool for agent-based modelling is NetLogo [79, 80]. This tool provides a complete development environment with its own programming language. The difference of NetLogo and our tool is that NetLogo's focus is modelling simple entities that move through their environment. The focus of our tool is to represent the decision making process of agents and their interaction with an environment as appropriately as possible. For example, the interaction between agents and environments is well defined in our formalism by influences and reactions. NetLogo's language does not provide such constructs. On the other hand, NetLogo has an extensive collection of built-in methods to move agents, determine their orientation, etc. This is not featured in our formalism. Another big difference is that NetLogo's semantics are not formal or precise. Our tool has precise semantics, which allows other groups to develop a tool that implements the same semantics.

A tool that is heavily inspired by NetLogo is ReLogo [65]. ReLogo is a logo dialect, which can be used to develop models in the Repast Symphony [23] framework. Due to the high similarity between ReLogo and NetLogo, the same discussion applies here.

Another tool that is inspired a lot by NetLogo is TurtleKit [57]. Like our tool, TurtleKit is implemented in an existing multi-agent platform, MadKit [39]. TurtleKit has made certain different design choices than we. TurtleKit is implemented as a layer on top of MadKit and provides its own Logo like concrete syntax. We reuse SARL's syntax for creating models. Also, the entity that is responsible for directing the activities within the model is the scheduler agent. This entity is an agent within the MadKit platform. We deliberately decided not to implement the simulator as an agent, but provided a simulator that has the role of a run time environment for the agents and environments.

A tool that is not part of the logo family is MASON [49]. MASON provides a Java based library for developing and simulating agent-based models. MASON provides a

set of classes that can be used as environment, additionally agents can be developed by extending the steppable entity which can be scheduled. MASON differs from our tool in that agents the structure for an agent in MASON is really simplistic. Again, this tool does not specify how agents interact with their environment, which is one the main focuses of our tool. Also no precise semantics of the tool were found.

6.3 Other agent-based formalisms

Apart from existing tools, there are multiple formalisms defined for agent-based modelling and simulation. We will list now an overview of a subset of these formalisms.

An action model that formalizes how agents and the environment affect the global state of the environment is the Influence-Reaction-Model for Simulation or IRM4S [56]. In this model, all entities first output their desires to perform an action or influences. Then the global state of the model is computed based on the complete set of influences. This model was a great influence for our proposed formalism. We incorporated this model by reusing parts of of the Parallel-DEVS formalism [92].

The behaviour of an agent can be implemented in many ways. The belief-desire-intention model [36] provides a high-level description for complex behaviour. Herein, the beliefs consist of the local information the agent has about itself and its environment. Desires or goals describe the desired end state of the system of an agent. Intention describe short term plans for achieving the goals. In Section 4.1.4, we discuss how the BDI model can be realized in our formalism.

In [82], Uhrmacher et al. describe how agents can be represented in JAMES, which is a discrete event modelling tool. JAMES's semantics are very similar to DEVS and the approach provided in the study can also be applied on the DEVS formalism. DEVS, however, is a very general formalism that is able to implement agents in many ways. Our formalism is heavily inspired by the DEVS formalism (and its extensions DS-DEVS[9], and Parallel-DEVS[92]). However, we try to provide a single structure for defining agents as well as models.

In [60], a formalism for agent-based modelling is presented based on the DEVS formalism, called M-DEVS. In this paper, it is shown that the IRM4S can be implemented with Parallel-DEVS. This was a very important influence for the design of our formalism. However, M-DEVS differs from our formalism in that it does not make a difference between agents and environments. Another difference is that we do not prohibit a time advance of 0. In M-DEVS, state transitions are divided into physical state transitions and logical state transitions. The physical state can not induce a time advance of 0, because instantaneous state transitions do not occur in real life. The logical state transitions and the logical output function is used to distribute information.

Another DEVS related formalism for agent-based modelling and simulation is LDEF

[8]. From LDEF, we reused the action model in our formalism to represent agents. However, LDEF does not implement the IRM4S, but implements a more traditional action model. In our formalism we updated the action model such that it implements the IRM4S. Another difference is that models in LDEF are hierarchical. Action models are grouped into agents, which are grouped into agent groups and so on. We decided to have a flat organizations, where an agent model comprises a set of agents and environments, but these are not grouped in any way. Organizational structures need to be defined as an environment.

CHAPTER 7

Conclusion

This thesis presents a possible solution for the lack of uniformity in the field of agent-based modeling and simulation, by presenting a formalism with precise semantics. Additionally, we show how this formalism is realized in a tool which allows modelers to model and simulate agent-based models in our formalism.

The contributions of this thesis include:

- An thorough analysis of field of agent-based modeling, were we discuss the essential features of an agent-based model and how they are realized in existing agent-based modeling tools and formalism.
- A formalism for agent-based modeling and simulation with precise semantics. The main purpose of our formalism focuses on the decision making process of the agents, as well as the interaction between agents and there environment, which is rarely seen in other tools. Instead of defining a completely new formalism, we synthesized our formalism from existing formalisms for agent-based modeling and simulation.
- A agent-based modeling tool that implements a formalism with precise semantics. We also performed a comparative study between our new tool and other existing agent-based modeling tools.

We conclude that it is not only feasible to create formalisms for agent-based modeling and simulation, but that it is a precondition for enabling future development of the field. The high complexity of agent-based models calls for a universal theory and designated methods to create and analyze these models. Our contributions are a step in this direction. A formalism with precise semantics allows the development of techniques that can be applied to all tools that implement the formalism. Moreover,

models can be reused and translated into different tools, with equivalent semantics. We hope to see future research into the unification of agent-based modeling and simulation.

Bibliography

[1] The one-of documentation of netlogo. `http://ccl.northwestern.edu/netlogo/docs/dict/one-of.html`. Accessed: 17-Januari-2020.

[2] University of antwerp. `http://www.uantwerpen.be/`. Accessed: 17-September-2019.

[3] Sameera Abar, Georgios K Theodoropoulos, Pierre Lemarinier, and Gregory MP O'Hare. Agent based modelling and simulation tools: a review of the state-of-art software. *Computer Science Review*, 24:13–33, 2017. Amount of References: 44.

[4] Gul Agha. Concurrent programming using actors. *Object-oriented concurrent programming*, pages 37–53, 1987.

[5] Gul Agha, Peter Wegner, and Akinori Yonezawa. *Research directions in concurrent object-oriented programming*. Mit Press, 1993.

[6] Gul A Agha and Wooyoung Kim. Actors: A unifying model for parallel and distributed computing. *Journal of systems architecture*, 45(15):1263–1277, 1999.

[7] Edouard Amouroux, Thanh-Quang Chu, Alain Boucher, and Alexis Drogoul. Gama: an environment for implementing and running spatially explicit multi-agent simulations. In *Pacific Rim International Conference on Multi-Agents*, pages 359–371. Springer, 2007.

[8] Jang Won Bae and Il-Chul Moon. Ldef formalism for agent-based model development. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 46(6):793–808, 2015.

[9] Fernando J Barros. Dynamic structure discrete event system specification: a new formalism for dynamic structure modeling and simulation. In *Winter Simulation Conference Proceedings, 1995.*, pages 781–785. IEEE, 1995.

[10] Maurice Stevenson Bartlett. Stochastic population models; in ecology and epidemiology. Technical report, 1960.

[11] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing multi-agent systems with JADE*, volume 7. John Wiley & Sons, 2007.

[12] Carole Bernon, Marie-Pierre Gleizes, Sylvain Peyruqueou, and Gauthier Picard. Adelfe: a methodology for adaptive multi-agent systems engineering. In *International Workshop on Engineering Societies in the Agents World*, pages 156–169. Springer, 2002.

[13] Paul-Antoine Bisgambiglia, Paul Antoine Bisgambiglia, and Romain Franceschini. Agent-oriented approach based on discrete event systems.

[14] Eric Bonabeau. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*, 99(suppl 3):7280–7287, 2002.

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

[16] Grady Booch. *Object oriented analysis & design with application.* Pearson Education India, 2006.

[17] Michael Bratman. *Intention, plans, and practical reason*, volume 10. Harvard University Press Cambridge, MA, 1987.

[18] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE journal on robotics and automation*, 2(1):14–23, 1986.

[19] Bruce G Buchanan and Tom M Mitchell. Model-directed learning of production rules. In *Pattern-directed inference systems*, pages 297–312. Elsevier, 1978.

[20] Stephanie Cammarata, David McArthur, and Randall Steeb. Strategies of cooperation in distributed problem solving. In *Readings in Distributed Artificial Intelligence*, pages 102–105. Elsevier, 1988.

[21] Davy Capera, J-P Georgé, M-P Gleizes, and Pierre Glize. The amas theory for complex problem solving based on self-organizing cooperative agents. In *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.*, pages 383–388. IEEE, 2003.

[22] Adam Cheyer and David Martin. The open agent architecture. *Autonomous Agents and Multi-Agent Systems*, 4(1):143–148, 2001.

[23] Nick Collier. Repast: An extensible framework for agent simulation. *The University of Chicago's Social Science Research*, 36:2003, 2003.

[24] Daniel D Corkill and Victor R Lesser. The use of meta-level control for coordination in a distributed problem solving network. Technical report, MAS-

SACHUSETTS UNIV AMHERST DEPT OF COMPUTER AND INFORMATION SCIENCE, 1983.

[25] Massimo Cossentino. From requirements to code with passi methodology. In *Agent-oriented methodologies*, pages 79–106. IGI Global, 2005.

[26] Robert G Coyle. System dynamics modelling: a practical approach. *Journal of the Operational Research Society*, 48(5):544–544, 1997.

[27] Elizabeth Donkin, Peter Dennis, Andrey Ustalakov, John Warren, and Amanda Clare. Replicating complex agent based models, a formidable task. *Environmental modelling & software*, 92:142–151, 2017.

[28] A. Dorri, S. S. Kanhere, and R. Jurdak. Multi-agent systems: A survey. *IEEE Access*, 6:28573–28593, 2018. Amount of References: 0.

[29] Edmund H Durfee. *Coordination of distributed problem solvers*, volume 55. Springer Science & Business Media, 2012.

[30] Oren Etzioni and Daniel S Weld. Intelligent agents on the internet: Fact, fiction, and forecast. *IEEE expert*, 10(4):44–49, 1995.

[31] Jacques Ferber, Olivier Gutknecht, and Fabien Michel. Agent/group/roles: Simulating with organizations. In *Fourth International Workshop on Agent-Based Simulation (ABS03)*, 2003.

[32] Jacques Ferber, Olivier Gutknecht, and Fabien Michel. From agents to organizations: an organizational view of multi-agent systems. In *International workshop on agent-oriented software engineering*, pages 214–230. Springer, 2003.

[33] Jacques Ferber and Gerhard Weiss. *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading, 1999. Amount of References: 3893.

[34] Michael R Genesereth and Nils J Nilsson. Logical foundations of. *Artificial Intelligence. New York: Morgan Kaufmann Publishers*, 1987.

[35] Michael R Genesereth and Nils J Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann, 2012.

[36] Michael Georgeff, Barney Pell, Martha Pollack, Milind Tambe, and Michael Wooldridge. The belief-desire-intention model of agency. In Jörg P. Müller, Anand S. Rao, and Munindar P. Singh, editors, *Intelligent Agents V: Agents Theories, Architectures, and Languages*, pages 1–10, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[37] Cláudio Gomes, Joachim Denil, and Hans Vangheluwe. Causal-block diagrams. 2016.

[38] Arnaud Grignard, Patrick Taillandier, Benoit Gaudou, Duc An Vo, Nghi Quang Huynh, and Alexis Drogoul. Gama 1.6: Advancing the art of complex agent-

based modeling and simulation. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 117–131. Springer, 2013.

[39] Olivier Gutknecht and Jacques Ferber. Madkit: A generic multi-agent platform. In *Proceedings of the Fourth International Conference on Autonomous Agents*, AGENTS '00, pages 78–79, New York, NY, USA, 2000. ACM.

[40] Aaron Helsinger, Michael Thome, and Todd Wright. Cougaar: a scalable, distributed multi-agent architecture. In *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No. 04CH37583)*, volume 2, pages 1910–1917. IEEE, 2004.

[41] Vincent Hilaire, Olivier Simonin, Abder Koukam, and Jacques Ferber. A formal approach to design and reuse agent and multiagent models. In James Odell, Paolo Giorgini, and Jörg P. Müller, editors, *Agent-Oriented Software Engineering V*, pages 142–157, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. Amount of References: 28.

[42] Nicholas R Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, 1(1):7–38, 1998.

[43] Nick Jennings and Michael Woolridge. Software agents. *IEE Review*, pages 17–20, 1996.

[44] Dennis Kafura and Jean-Pierre Briot. Actors and agents. *IEEE Concurrency*, (2):24–28, 1998.

[45] David Kotz and Robert S Gray. Mobile agents and the future of the internet. *ACM Operating Systems Review*, 1999.

[46] Paulo Leitao, Stamatis Karnouskos, Luis Ribeiro, Jay Lee, Thomas Strasser, and Armando W Colombo. Smart agents in industrial cyber–physical systems. *Proceedings of the IEEE*, 104(5):1086–1101, 2016.

[47] Tim Leys. The current state of agent-based modeling. 2019.

[48] Jing Lin, Sahra Sedigh, and Ann Miller. Modeling cyber-physical systems with semantic agents. In *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*, pages 13–18. IEEE, 2010. Amount of References: 66.

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

[50] Charles Macal and Michael North. Introductory tutorial: Agent-based modeling and simulation. In *Proceedings of the 2014 winter simulation conference*, pages 6–20. IEEE Press, 2014.

[51] Charles M Macal. Everything you need to know about agent-based modelling and simulation. *Journal of Simulation*, 10(2):144–156, 2016.

[52] Charles M Macal and Michael J North. Tutorial on agent-based modelling and simulation. *Journal of simulation*, 4(3):151–162, 2010. Amount of References: 1235.

[53] P Maes. [a]agents that reduce work and information overload, [o] comm, 1994.

[54] Fabien Michel. Introduction to turtlekit: A platform for building logo based multi-agent simulations with madkit. 2002.

[55] Fabien Michel. *Formalisme, outils et éléments méthodologiques pour la modélisation et la simulation multi-agents*. PhD thesis, 2004.

[56] Fabien Michel. The irm4s model: the influence/reaction principle for multiagent based simulation. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 133. ACM, 2007.

[57] Fabien Michel, Grégory Beurier, and Jacques Ferber. The turtlekit simulation platform: Application to complex systems. In *SITIS: Signal-Image Technology and Internet–Based Systems*, 2005.

[58] Fabien Michel, Jacques Ferber, and Alexis Drogoul. Multi-agent systems and simulation: A survey from the agent commu-nity's perspective. In *Multi-Agent Systems*, pages 17–66. CRC Press, 2018.

[59] Tom M Mitchell. Machine learning, 1997.

[60] Jean-Pierre Müller. Towards a formal semantics of event-based multi-agent simulations. In *International Workshop on Multi-Agent Systems and Agent-Based Simulation*, pages 110–126. Springer, 2008.

[61] Nicola Muscettola, P Pandurang Nayak, Barney Pell, and Brian C Williams. Remote agent: To boldly go where no ai system has gone before. *Artificial intelligence*, 103(1-2):5–47, 1998.

[62] Nils J Nilsson. Shakey the robot. Technical report, SRI INTERNATIONAL MENLO PARK CA, 1984.

[63] James J Odell, H Van Dyke Parunak, Mitch Fleischer, and Sven Brueckner. Modeling agents and their environment. In *International Workshop on Agent-Oriented Software Engineering*, pages 16–31. Springer, 2002.

[64] Guy H Orcutt. A new type of socio-economic system. *The review of economics and statistics*, pages 116–123, 1957.

[65] Jonathan Ozik, Nicholson T Collier, John T Murphy, and Michael J North. The relogo agent-based modeling language. In *2013 Winter Simulations Conference (WSC)*, pages 1560–1568. IEEE, 2013.

[66] H Van Dyke Parunak. " go to the ant": Engineering principles from natural multi-agent systems. *Annals of Operations Research*, 75:69–101, 1997.

[67] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[68] Gauthier Picard and Marie-Pierre Gleizes. The adelfe methodology. In *Methodologies and Software Engineering for Agent Systems*, pages 157–175. Springer, 2004.

[69] Mark Pinsky and Samuel Karlin. *An introduction to stochastic modeling*. Academic press, 2010.

[70] Steven F Railsback, Steven L Lytinen, and Stephen K Jackson. Agent-based simulation platforms: Review and development recommendations. *Simulation*, 82(9):609–623, 2006.

[71] Alessandro Ricci, Mirko Viroli, and Michele Piunti. Formalising the environment in mas programming: A formal model for artifact-based environments. In *International Workshop on Programming Multi-Agent Systems*, pages 133–150. Springer, 2009.

[72] Sebastian Rodriguez, Nicolas Gaud, and Stéphane Galland. Sarl: a general-purpose agent-oriented programming language. In *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, volume 3, pages 103–110. IEEE, 2014.

[73] Stanley J Rosenschein. Formal theories of knowledge in ai and robotics. *New generation computing*, 3(4):345–357, 1985.

[74] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 3th edition, 2016.

[75] Yoav Shoham. Agent-oriented programming. *Artificial intelligence*, 60(1):51–92, 1993.

[76] Burrhus Frederic Skinner. *Science and human behavior*. Number 92904. Simon and Schuster, 1953.

[77] Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.

[78] Eric Tatara, M North, T Howe, N Collier, Jerry Vos, et al. An indroduction to repast simphony modeling using a simple predator-prey example. In *Proceedings of the Agent 2006 Conference on Social Agents: Results and Prospects*, 2006.

[79] Seth Tisue and Uri Wilensky. Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, volume 21, pages 16–21. Boston, MA, 2004.

[80] Seth Tisue and Uri Wilensky. Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, volume 21, pages 16–21. Boston, MA, 2004. Amount of References: 566.

[81] Adelinde M Uhrmacher. Dynamic structures in modeling and simulation: a reflective approach. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 11(2):206–232, 2001.

[82] Adelinde M Uhrmacher and Bernd Schattenberg. Agents in discrete event simulation. In *European Simulation Symposium-ESS*, volume 98, pages 129–136, 1998.

[83] Adelinde M Uhrmacher and Danny Weyns. *Multi-Agent systems: Simulation and applications*. CRC press, 2009.

[84] Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, and Hans Vangheluwe. Domain-specific modelling for human–computer interaction. In *The Handbook of Formal Methods in Human-Computer Interaction*, pages 435–463. Springer, 2017.

[85] Yentl Van Tendeloo and Hans Vangheluwe. An introduction to classic devs. *arXiv preprint arXiv:1701.07697*, 2017.

[86] José M Vidal, Paul A Buhler, and Michael N Huhns. Inside an agent. *IEEE Internet Computing*, 5(1):82–86, 2001.

[87] Vito Volterra. Fluctuations in the abundance of a species considered mathematically, 1926.

[88] Vito Volterra. Variations and fluctuations of the number of individuals in animal species living together. *Animal ecology*, pages 409–448, 1926.

[89] Shiyong Wang, Jiafu Wan, Daqiang Zhang, Di Li, and Chunhua Zhang. Towards smart factory for industry 4.0: a self-organized multi-agent system with big data based feedback and coordination. *Computer Networks*, 101:158–168, 2016.

[90] Gerhard Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press, 1999. Amount of References: 5368.

[91] Michael Wooldridgey and Paolo Ciancarini. Agent-oriented software engineering: The state of the art. In *International Workshop on Agent-Oriented Software Engineering*, pages 1–28. Springer, 2000.

[92] Bernard P Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of modeling and simulation*. Academic press, 2000.

[93] Bernard P Zeigler and Sankait Vahie. Devs formalism and methodology: unity of conception/diversity of application. In *Proceedings of 1993 Winter Simulation Conference-(WSC'93)*, pages 573–579. IEEE, 1993.

Appendices

APPENDIX A

Scenario Output Traces

In this appendix, all ouput traces from the test scenarios are listed.

A.1 Output trace from the second traffic light scenario

```
Simulating Timestep 7.0
Elapsed time: 7.0
Current color: Red
```

```
Simulating Timestep 24.0
Elapsed time: 17.0
Current color: Orange
```

```
Simulating Timestep 31.0
Elapsed time: 7.0
Current color: Red
```

```
Simulating Timestep 35.0
Elapsed time: 4.0
Current color: Green
```

```
Simulating Timestep 59.0
Elapsed time: 24.0
Current color: Orange
```

```
Simulating Timestep 66.0
Elapsed time: 7.0
Current color: Red
```

```
Simulating Timestep 83.0
Elapsed time: 17.0
Current color: Orange
```

```
Simulating Timestep 90.0
Elapsed time: 7.0
Current color: Red
```

```
Simulating Timestep 94.0
Elapsed time: 4.0
Current color: Green
```

```
Simulating Timestep 118.0
Elapsed time: 24.0
Current color: Orange
```

A.2 Output trace reactive agent scenario

```
Simulating Timestep 1.0
[PingAgent] Send Ping
```

```
Simulating Timestep 1.0
[PongAgent] Received Ping
```

```
Simulating Timestep 2.0
[PongAgent] Send Pong
```

```
Simulating Timestep 2.0
[PingAgent] Received Pong
```

```
Simulating Timestep 3.0
[PingAgent] Send Ping
```

```
Simulating Timestep 3.0
[PongAgent] Received Ping
```

```
Simulating Timestep 4.0
```

```
[PongAgent] Send Pong
```

```
Simulating Timestep 4.0
[PingAgent] Received Pong
```

```
Simulating Timestep 5.0
[PingAgent] Send Ping
```

```
Simulating Timestep 5.0
[PongAgent] Received Ping
```

A.3 Output trace dynamic population scenario

```
New agent started
```

```
Simulating Timestep 5.0
New agent started
```

```
Simulating Timestep 5.0
```

```
Simulating Timestep 10.0
New agent started
New agent started
```

```
Simulating Timestep 10.0
```

```
Simulating Timestep 15.0
New agent started
New agent started
New agent started
New agent started
```

```
Simulating Timestep 15.0
```

```
Simulating Timestep 20.0
New agent started
New agent started
New agent started
New agent started
New agent started
New agent started
New agent started
New agent started
```

A.4 Output trace single car scenario

```
Simulating Timestep 1.0
Agent with ID: 00000000-0000-0001-0000-000000000000
speed: 0.8228709999999999
position: 0.8228709999999999
```

```
Simulating Timestep 2.0
Agent with ID: 00000000-0000-0001-0000-000000000000
speed: 0.8878709999999999
position: 1.7107419999999998
```

```
Simulating Timestep 3.0
Agent with ID: 00000000-0000-0001-0000-000000000000
speed: 0.9528709999999998
position: 2.663613
```

```
Simulating Timestep 4.0
Agent with ID: 00000000-0000-0001-0000-000000000000
speed: 1.0
position: 3.663613
```

```
Simulating Timestep 5.0
Agent with ID: 00000000-0000-0001-0000-000000000000
speed: 1.0
position: 4.663613
```

```
Simulating Timestep 6.0
Agent with ID: 00000000-0000-0001-0000-000000000000
speed: 1.0
position: 0.6636129999999998
```