UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FERNANDO DOS SANTOS

Model-Driven Agent-based Simulation Development

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dra. Ana L. C. Bazzan
Coadvisor: Prof. Dra. Ingrid Nunes

Porto Alegre
January 2019

CIP — CATALOGING-IN-PUBLICATION

Santos, Fernando dos

Model-Driven Agent-based Simulation Development / Fernando dos Santos. – Porto Alegre: PPGC da UFRGS, 2019.

145 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2019. Advisor: Ana L. C. Bazzan; Coadvisor: Ingrid Nunes.

1. Agent-based Modeling and Simulation. 2. Model-driven Development. 3. Domain-specific Modeling Language. 4. User Study. I. Bazzan, Ana L. C.. II. Nunes, Ingrid. III. Título.

ABSTRACT

The Agent-based Modeling and Simulation (ABMS) paradigm has been used to analyze, reproduce, and predict phenomena in many application areas, such as traffic and epidemiology. Building agent-based simulations is a challenging task that often demands technical expertise in ABMS and its simulation platforms. Researchers have already argued about the importance of tools and building blocks that increase the abstraction level and therefore reduce the effort in agent-based simulation development. Model-driven Development (MDD) is an approach for software development, in which high-level modeling artifacts drive the production of effort-consuming low-level artifacts, such as the source code. Previous studies on the use of MDD in mainstream software development have already shown that it significantly increases productivity. However, in the ABMS paradigm MDD has been exploited in a limited way. Most of the existing proposals consider modeling and code generation of limited simulation aspects, leaving much left to be implemented by developers. Nevertheless, there is a lack of empirical studies that demonstrate whether these MDD approaches are indeed effective. In this thesis, we exploit MDD in the context of ABMS. We propose MDD4ABMS, a model-driven approach for developing agent-based simulations. MDD4ABMS is composed of the following elements, which are the main contributions of this thesis: (i) a metamodel for agent-based simulations that was built following a bottom-up approach to abstract aspects recurrently used in simulations and make them available for modeling; (ii) a domain-specific modeling language with building blocks to instantiate agent-based simulation models; and (iii) model-to-code transformations to generate source code for NetLogo, a widely used agent-based simulation platform. While abstractions provided by the metamodel allow developers to focus on *which* features to consider in simulations instead of *how* to implement and integrate them (a task that may introduce inconsistencies in simulations), building blocks provided by the modeling language promote expressive modeling. Empirical studies showed that MDD4ABMS reduces the development effort in comparison to NetLogo, and meets qualitative aspects related to the user experience, such as ease of comprehension and usability. These results give evidence of the benefits that MDD4ABMS provide to ABMS.

Keywords: Agent-based Modeling and Simulation. Model-driven Development. Domain-specific Modeling Language. User Study.

Desenvolvimento Dirigido a Modelos de Simulações baseadas em Agentes

RESUMO

O paradigma de modelagem e simulação baseadas em agentes (do inglês, ABMS) tem sido usado para analisar, reproduzir e predizer fenômenos em diversas áreas, tais como tráfego e epidemiologia. A construção de simulações baseadas em agentes é uma tarefa desafiadora que frequentemente demanda conhecimento em ABMS e suas plataformas de simulação. Pesquisadores já destacaram a importância de ferramentas e blocos de construção que aumentem o nível de abstração e consequentemente reduzam o esforço de desenvolvimento de simulações baseadas em agentes. Desenvolvimento dirigido a modelos (do inglês, MDD) é uma abordagem para desenvolvimento de software em que artefatos de modelagem de alto nível conduzem a produção dos artefatos de mais baixo nível que demandam significativo esforço, como por exemplo a produção de código fonte. No desenvolvimento de software convencional, estudos mostram que o uso de MDD aumenta significativamente a produtividade. Entretanto, no paradigma ABMS, o uso de MDD tem sido explorado de forma limitada. Na maioria das propostas existentes, a modelagem e geração de código é limitada a alguns aspectos, logo, muito ainda precisa ser implementado pelos desenvolvedores. Além disto, há uma carência de estudos empíricos que demonstrem se tais propostas são de fato efetivas. Nesta tese exploramos MDD no contexto de ABMS. Propomos MDD4ABMS, uma abordagem dirigida a modelos para desenvolvimento de simulações baseadas em agentes. MDD4ABMS é composto dos seguintes elementos, que são as principais constribuições desta tese: (i) um metamodelo para simulações baseadas em agentes, construído de forma *bottom-up* a fim de abstrair os aspectos recorrentemente usados em simulações e disponibilizá-los para uso em modelos; (ii) uma linguagem de modelagem, que oferece blocos de construção para instanciar modelos de simulações baseadas em agentes; e (iii) transformações modelo-para-código, que geram código fonte para NetLogo, uma plataforma de simulação largamente utilizada. Enquanto as abstrações providas pelo metamodelo permitem aos desenvolvedores focar em *quais* aspectos desejam considerar nas simulações em vez de *como* implementá-los e integrá-los (uma tarefa que pode introduzir inconsistências nas simulações), os blocos de construção providos pela linguagem promovem modelagem expressiva. Estudos empíricos mostraram que MDD4ABMS reduz o esforço de desenvolvimento em comparação ao NetLogo, e satisfaz aspectos qualitativos relacionados à experiência do usuário, tais

como facilidade de compreensão e usabilidade. Tais resultados fornecem evidência dos benefícios que MDD4ABMS oferece para ABMS.

Palavras-chave: Modelagem e Simulação baseada em Agentes, Desenvolvimento Dirigido a Modelos, Linguagem de Modelagem Específica de Domínio, Estudo com Usuários.

LIST OF ABBREVIATIONS AND ACRONYMS

ABMS	Agent-based Modeling and Simulation
AI	Artificial Intelligence
AME	Atomic Model Element
AOSE	Agent-Oriented Software Engineering
ATSC	Adaptive Traffic Signal Control
BDI	Beliefs, Desires, and Intentions
BPMN	Business Process Modeling Notation
CSV	Comma Separated Values
DEVS	Discrete Event System Specification
DSL	Domain-specific Language
FQAD	Framework for Qualitative Assessment of DSLs
GIS	Geographic Information System
GQM	Goal-Question-Metric
LOC	Line of Code
MAS	Multiagent System
MDD	Model-driven Development
ODD	Overview, Design concepts, and Details
OOP	Object Oriented Programming
OSM	Open Stree Map
TSC	Traffic Signal Controller
UML	Unified Modeling Language

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Agent-based simulations have been widely used to understand the emergent behavior of complex systems. These systems are composed of multiple entities, or agents, which can interact with each other and are situated in an environment that they can perceive and modify through their actions. It is not trivial to formulate analytical models that can simulate the behavior of such complex systems. Building them is a challenging task that has been widely investigated in the context of Agent-based Modeling and Simulation (ABMS), a simulation paradigm that uses autonomous agents and multiagent systems to reproduce and explore a phenomenon under investigation.

The ABMS paradigm has been used to simulate systems in many application areas, such as traffic, ecology, economics, and epidemiology (MACAL; NORTH, 2014). According to Macal and North (2014), ABMS is selected as the simulation paradigm in such areas because it can explicitly incorporate the complexity arising from individual behavior and interactions that exist in real settings. Moreover, the closer correspondence between agent models and the target system reduces the modeling gap and hence ABMS is a step towards accuracy and precision in modeling (EDMONDS, 2001).

An agent-based model must explicitly deal with the following three elements: *agents*, *interactions* and *environment*. An agent-based simulation system (BANDINI; MANZONI; VIZZARI, 2009) is a software system composed of an agent-based model and an execution specification. When a simulation runs, processes associated with the agent-based model are executed repeatedly over the timeline defined in the execution specification. Additionally, such specification also determines the temporal scale and extent, how many agents are created and their initial states, and which data is collected for further analysis.

As in the case of any software system, the development of an agent-based simulation system is a process that often involves different roles, from experts of an application area to developers (GALÁN et al., 2009). From the conceptualization, goals, and purpose of the simulation produced by application area experts, developers build the simulation system. Though mainstream software developers have technical expertise in programming logic, expertise in the ABMS paradigm may be an issue, which may affect the design quality of simulations, as well as productivity, even for experienced mainstream software developers.

The ABMS community has been working to provide alternatives for developing

agent-based simulation systems. On the one hand, methodologies, processes, and metamodels have been proposed (GARRO; RUSSO, 2010; KLÜGL; DAVIDSSON, 2013; GHORBANI et al., 2013). Although these alternatives provide support for specifying most aspects of the agent-based model, support for specifying execution aspects is limited, often absent. On the other hand, there are simulation platforms such as NetLogo and Repast. These platforms provide programming environments with language constructs to define agents, interactions, and the environment, as well as execution aspects. However, previous expertise in programming logic is required to implement the simulation dynamics.

Although agent-based simulation systems have been developed using the existing alternatives, researchers have already argued about the importance of providing building blocks and tools at a higher abstraction level (PARUNAK; SAVIT; RIOLO, 1998; HAMILL, 2010; WELLMAN, 2016). As pointed out by Macal (2016, p. 152), "*lack of easy-to-use tools and standardized user interfaces for ABMS model development is a barrier to adoption of ABMS*." Such approaches already exist in other contexts. For example, Simulink offers abstractions related to electric motors and power control to simulate dynamic systems, and the Business Process Modeling Notation (BPMN) provides abstractions related to activities and participants to model business processes. These approaches provide languages tailored for expressive modeling, reducing the abstraction gap in models. In the context of ABMS, such approaches would potentially ease the development of simulation systems and increase productivity.

In agent-based simulations, agents can be endowed with learning or evolutionary capabilities to adapt to changes in themselves or the environment. Artificial Intelligence (AI) techniques that provide such capabilities are well established and can be incorporated into agents leading to more realistic simulations (KLÜGL; BAZZAN, 2012). However, such agent capabilities are usually developed from scratch in existing approaches for ABMS. Consequently, besides determining *which* features are considered in a simulation (e.g., which learning technique or spread of disease model is adopted by agents), the designer must also know *how* to implement and integrate them with other simulation elements. Incorrect implementation of these features may introduce inconsistencies in the simulation. In the particular case of agent-based simulations, where the output is usually unknown beforehand, these inconsistencies may lead to wrong findings. Detecting and fixing such inconsistencies is a complex and effort consuming task that affects productivity and may frustrate ABMS beginners. Therefore, providing the simula-

tion designer with reusable models of these features from which error-free source code is generated would facilitate the development of simulation systems and foster the adoption of the ABMS paradigm.

Given this context, in this thesis we investigate expressiveness and productivity in agent-based simulation development. In Section 1.1 we present the problem we address in this thesis and the limitations of existing work. In Section 1.2 we describe our proposed solution and provide an overview of the contributions.

1.1 Problem Statement and Limitations of Existing Work

As introduced earlier, in this thesis we aim to tackle two issues of the available alternatives for ABMS: limited expressiveness and productivity. Based on these two issues, the research question considered in this thesis is stated as follows.

Research Question.

How to reduce the effort in agent-based simulation development with mainstream software developers that have little expertise in ABMS?

Limitations of existing work, which are associated with this research question, and issues associated with agent-based simulation development are as follows.

(1) **Limited expressiveness.** Expressiveness is related to the ability to represent an agent model and its execution specification with high-level, ABMS-related, abstractions (i.e., abstractions of recurrent agent capabilities used in simulations, or recurrent topologies for the simulated environment). Such abstractions would provide building blocks, reduce the abstraction gap, and increase productivity.

An evaluation of existing Agent-Oriented Software Engineering (AOSE) methodologies, processes, and metamodels, as well as platforms focused on agent-based simulation development, showed that they support the specification of limited aspects of an agent model and its execution specification. Moreover, building blocks for aspects recurrently specified in models are rarely provided by these approaches. On the one hand, there are approaches that rely on the UML for modeling agent-based simulation systems—e.g., MDA4ABMS (GARRO; PARISI; RUSSO, 2013), ODiM (DUARTE; LARA, 2009), and the approach by Iba, Matsuzawa and Aoyama

(2004). However, Bauer and Odell (2005) already reported that the UML does not provide expressive constructs to specify sophisticated aspects of agent-based simulations such as learning and adaptation. On the other hand, approaches such as AMASON (KLÜGL; DAVIDSSON, 2013) and MAIA (GHORBANI et al., 2013) do not rely on the UML. However, these approaches abstract only basic aspects of agent models, and sophisticated aspects are left out as well. Finally, existing platforms for agent-based simulation development provide limited abstractions for aspects recurrently used in simulations, as detailed next.

(2) **Demand for expertise in ABMS and, in particular, in models and techniques adopted (or desired) in simulations.** Although existing approaches for developing agent-based simulation systems, in particular platforms for simulation development, already do the job (i.e., one can develop and run fully-featured simulations by using them), they provide limited building blocks for aspects recurrently used in simulations. Developers of agent-based simulation systems must usually specify and implement these recurrent aspects from scratch. Hence, besides modeling or programming skills in the selected approach, those developers must have expertise in models and techniques adopted in simulations. For example, to develop a spread of disease simulation, a developer must know which epidemiological model must be used and how to implement it. As mentioned earlier, developing these elements from scratch compromises productivity and can introduce inconsistencies in simulations.

(3) **Lack of assessment of the effectiveness of existing approaches involving development tasks performed by developers.** Despite qualitative arguments that motivate the use of existing alternatives for developing agent-based simulation systems, there is a need for concrete evidence that they, in fact, promote expressiveness and productivity. Usually, only examples and case studies that demonstrate the *feasibility* of using these alternatives to develop simulation systems are available. No *effectiveness* evaluations are presented to show that these alternatives indeed provide benefits to mainstream software developers with little expertise in ABMS regarding expressiveness and productivity.

1.2 Proposed Solution and Contributions Overview

Software engineering supports mainstream software development by providing techniques for the specification, design, implementation, and evolution of systems. One of these techniques is Model-driven Development (MDD) (ATKINSON; KÜHNE, 2003; SCHMIDT, 2006). MDD aims to find domain-specific abstractions and make them available for modeling. In MDD models are thus considered *first-class citizens* and the development is driven by modeling artifacts (STAHL et al., 2006a).

An MDD approach combines Domain-specific Languages (DSLs) with model transformations. These are tied by a metamodel that specifies concepts and relationships among them within a domain. A model is thus an instance of such a metamodel. DSLs are modeling or programming languages tailored for a particular domain. The specific focus on a particular domain allows DSLs to express the key aspects of that domain, trading *generality* for *expressiveness* (STAHL et al., 2006a). Model transformations, in turn, allow changing the model abstraction level, as well as generating source code. When automated, such model transformations can be executed to produce source code automatically from models, which increases productivity and enhance quality in software development.

Considering that, in MDD approaches, domain-specific aspects are made available for modeling, it is a potential approach to promote expressiveness and productivity in agent-based simulation development. Previous work on the use of MDD in industry has already shown that MDD approaches increase productivity because the modeling effort is focused on domain concerns instead of programming statements (SPRINKLE et al., 2009; TOLVANEN; KELLY, 2016).

MDD approaches focused on ABMS have been proposed, such as MDA4ABMS (GARRO; PARISI; RUSSO, 2013) and ODiM (DUARTE; LARA, 2009), as well as the approaches by Iba, Matsuzawa and Aoyama (2004) and Gómez-Sanz, Fernández and Arroyo (2010). However, these approaches usually cover only basic aspects of simulations and have issues concerning expressiveness and code generation. Nevertheless, these MDD approaches have been mostly evaluated from the point of view of *feasibility*. Evaluation, with quantification, of *effectiveness* is almost never considered.

We thus investigate the following research hypothesis in this thesis.

Research Hypothesis.

With a model-driven approach focused on providing high-level abstractions for simulation aspects, expressive modeling, and source code generation, mainstream software developers with little expertise in ABMS are able to model and execute simulations with reduced effort.

We propose a model-driven approach for ABMS, called MDD4ABMS, and evaluate it to assess the benefits provided to mainstream software developers. MDD4ABMS is composed of (i) a metamodel for agent-based simulations, (ii) a domain-specific modeling language with tool support, and (iii) model transformations for code generation. These three elements, as well as (iv) their empirical evaluation, are the main contributions of this thesis. Next, we briefly describe these main contributions, as well as other minor contributions.

(i) A **metamodel for agent-based simulations**, which abstracts aspects frequently used in simulations and make them available for modeling, addressing limitations (1) and (2) of existing work. It was build following a generative, bottom-up approach, which considered as input existing agent-based simulations. The metamodel is composed of a core (SANTOS; NUNES; BAZZAN, 2017c) and extensions for two application domains: adaptive traffic signal control (SANTOS; NUNES; BAZZAN, 2017b; SANTOS; NUNES; BAZZAN, 2017a; SANTOS; NUNES; BAZZAN, 2018), and spread of disease.

(ii) A **domain analysis method**, which allows extending the metamodel with abstractions from other application domains (SANTOS; NUNES; BAZZAN, 2018). When applied to a set of existing agent-based simulations it produces as output a list of abstractions that correspond to the domain model, which can be then incorporated into the metamodel.

(iii) A **domain-specific modeling language** DSL, called ABStractLang, which provides a more productive and expressive way for specifying agent-based simulations (SANTOS; NUNES; BAZZAN, 2017c; SANTOS; NUNES; BAZZAN, 2018). It is a modeling language that adopts graphical notation and provides building blocks to instantiate elements of the MDD4ABMS metamodel, contributing for addressing limitations (1) and (2) of existing work.

(iv) **Model transformations for code generation**, which generate code for NetLogo, a widely used agent-based simulation platform (SANTOS; NUNES; BAZZAN, 2018). This contribution addresses limitation (2) of existing work, in particular, productivity and consistency issues.

(v) A **modeling tool**, called ABStractme, which enable mainstream software developers to create agent-based simulation models using the ABStractLang modeling language (MOREIRA et al., 2017). ABStractme provides a diagram editor, and automatically executes the model-to-code transformations for generating source code for the simulation.

(vi) Three **empirical studies**, which evaluate different aspects of MDD4ABMS (SANTOS; NUNES; BAZZAN, 2017c; SANTOS; NUNES; BAZZAN, 2018). Results showed that MDD4ABMS effectively provide benefits for mainstream software developers with respect to design quality and development effort. Additionally, by conducting these studies we ensure that MDD4ABMS is not subject to limitation (3) of existing work.

MDD, as a research field, proposes solutions to a number of problems faced in software development. Model-to-model transformations allow model conversion between distinct software platforms or distinct programming paradigms. Round-trip engineering can be used to synchronize these models. However, as we were interested primarily in expressive modeling and productivity in simulation development, these aspects are not in the scope of this thesis.

1.3 Thesis Outline

The remainder of this thesis is organized into five chapters, as follows.

Chapter 2 presents the background on topics relevant to this thesis, describes related work and discusses their limitations.

Chapter 3 describes the MDD4ABMS metamodel, which abstracts recurrent aspects of agent-based simulations.

Chapter 4 details the domain-specific language ABStractLang created for modeling agent-based simulations, in addition to model-to-code transformations and the ABStractme modeling tool.

Chapter 5 presents the three empirical studies conducted to evaluate the benefits of the MDD4ABMS approach with respect to design quality, development effort, and subjective aspects such as ease of comprehension and usability.

Chapter 6 concludes this thesis, outlining directions for future work.

1.4 Disclaimer

Note about the adopted agent-based simulation terminology. Different terms are found in the literature to refer to the agent-based simulation paradigm. Agent-based Social Simulation (ABSS), Multiagent-based Simulation (MABS), Agent-based Simulation (ABS), and Agent-based Modeling and Simulation (ABMS) are among the terms most frequently adopted to refer to such paradigm. In this thesis, we follow the terminology used by Klügl and Bazzan (2012) and Macal and North (2010), and therefore we adopt ABMS to refer to the agent-based simulation paradigm. Additionally, the terms "agent-based simulation system" and "agent-based simulation" are used interchangeably to refer to a software system that realizes an agent-based model and thus can be used for running simulations.

Expected reader's background This thesis assumes that the reader has some degree of familiarity with multiagent systems and agent-based simulations. Although Chapter 2 presents some background on these subjects, those unfamiliar with agents and their reasoning cycle may find it difficult to comprehend the agent-based simulation metamodel proposed in this thesis and its assumptions. Newcomers in agent-based simulations are referred to Macal and North (2010), Klügl and Bazzan (2012), and Wilensky and Rand (2015) to get familiar with the topic.

Note about the semantic of the metamodel elements. As previously explained, the agent-based simulation metamodel proposed in this thesis captures aspects frequently used in simulations. The terminology adopted in the metamodel is straightly related to the abstracted aspects. For example, an agent is abstracted as an *agent* element, and its mobility is abstracted as an *mobility* element. Given this closely related correspondence between simulation aspects and their abstractions, the semantic (meaning) of the metamodel elements is not described in this thesis, which reinforces the need for the reader having got familiar with agent-based simulation terms. Constraints regarding valid relationships between metamodel elements are described throughout the text of this thesis.

2 BACKGROUND AND RELATED WORK

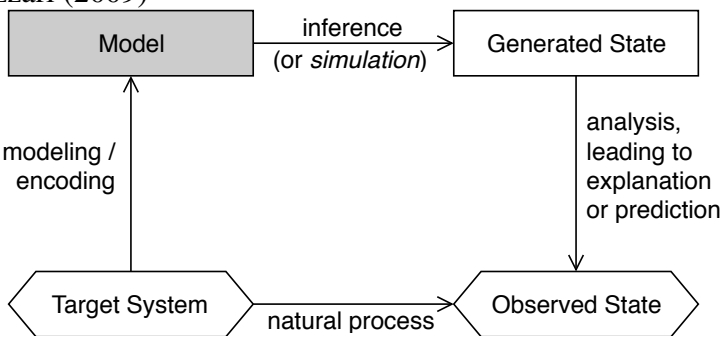
This chapter presents a brief overview of the topics relevant to this thesis. Section 2.1 focuses on the foundations and applications of agent-based modeling and simulation. Model-driven development and its elements are presented in Section 2.2. Section 2.3 presents related work on alternatives for developing agent-based simulations and discusses how they ease the development of simulations and promote development productivity. Finally, Section 2.4 points out remarks on these topics.

2.1 Agent-based Modeling and Simulation

2.1.1 Foundations

A model is an abstract (and often simplified) representation of a target system. Models can be specified for either existing or planned systems to enable predicting the state of the system through an inferential process (EDMONDS, 2001; BANDINI; MANZONI; VIZZARI, 2009). Figure 2.1 illustrates the relation between a model and its target system. According to Edmonds (2001), something is a model of something else if the diagram commutes, i.e., if the generated state of the model coherently explains or predicts the state observed from the natural process of the target system. Modeling and the inference produced from models are therefore predictive and explanatory instruments to gain additional insights about the target system (BANDINI; MANZONI; VIZZARI, 2009). Other reasons for modeling include straight replication, sensitivity analysis, to guide data collection, and to suggest analogies (EPSTEIN, 2008).

Figure 2.1: General Modeling Schema. Adapted from Edmonds (2001) and Bandini, Manzoni and Vizzari (2009)



Many target systems are characterized by the presence of autonomous entities, whose behaviors (actions and interactions) determine the evolution of the overall system. These systems can be modeled as Multiagent System (MAS), and their states can be generated by simulation (execution) (BANDINI; MANZONI; VIZZARI, 2009). Building these agent systems to allow inference through simulation is the focus of Agent-based Modeling and Simulation (ABMS), a simulation paradigm that uses autonomous agents and multiagent systems to analyze, reproduce, or predict phenomena.

A MAS is a system composed of agents. The commonly adopted definition of agent specifies the following set of properties that characterize an agent entity: (i) *autonomy*: the ability to operate without intervention; (ii) *reactivity*: the ability to perceive the environment in which it is situated and respond to changes on it; (iii) *pro-activeness*: the ability to take the initiative according its internal goals; and (iv) *social ability*: the possibility to interact with other agents (WOOLDRIDGE; JENNINGS, 1995).

From a multiagent view, each agent in the system can have *distinct abilities* or *architectures*, leading to heterogeneous systems. Social ability plays a key role when agents have only a limited perception of the situation and must interact to achieve their individual or collective goals. Considering these features of MAS, the ABMS paradigm supports the study and analysis of decentralized decision making, local-global interactions, self-organization, emergence, and the effects of heterogeneity in the modeled system (BANDINI; MANZONI; VIZZARI, 2009).

The following sequence of steps was presented by Edmonds (2001) for modeling and simulation in the ABMS paradigm.

1. **Abstraction**. The target system is abstracted, i.e., its relevant aspects are identified to build an analogical model of the system.
2. **Design**. A MAS is designed so as to incorporate that aspects previously identified.
3. **Inference**. The MAS is run (executed) to generate the model outputs.
4. **Analysis**. Model outputs are analyzed by a variety of means, which can include simple inspection, visualization techniques, or statistics. Such analysis verifies whether the model indeed represents the target system and validates whether results are acceptable in terms of outcomes.
5. **Interpretation**. Conclusions regarding the target system are drawn by interpreting the behavior displayed in the MAS.
6. **Application**. The knowledge gathered from the MAS is applied to explain or predict the target system.

This thesis is focused on the first three steps of this process, in which details about the structure of the target system and how it is going to be simulated are specified. Analysis, interpretation, and application follow from the model outputs. Those steps use well-known methods, such as statistical analysis, and they rely on the modeler introspection.

In the first two steps, a model of the target system is conceptualized in terms of multiagent systems. This model must explicitly deal with the following three elements: *agents*, *interactions* and *environment*. These are detailed next.

An *agent* is composed of properties (or attributes) and behaviors for enabling autonomy, reactivity, pro-activeness and social ability. According to Bandini, Manzoni and Vizzari (2009), agent behaviors are composed of two elements: (i) actions, which can cause modifications in the environment or in other agents; and (ii) a mechanism for deliberation, i.e., for selecting actions to be carried out according to the agent' perceptions and its internal state. Deliberation is related to the agent architecture (e.g., BDI—Beliefs, Desires, and Intentions), but is also related to agents' abilities such as learning and adaptation (BANDINI; MANZONI; VIZZARI, 2009; MACAL; NORTH, 2014).

The *environment* contains any other elements not modeled as agents, such as resources and objects. According to Weyns, Omicini and Odell (2007, p. 15), "*The environment is a first-class abstraction that provides the surrounding conditions for agents to exist and that mediates both the interaction among agents and the access to resources.*" Weyns, Omicini and Odell (2007) also states that the environment (i) is responsible for structuring the MAS physically (topology); (ii) embeds resources and services and supports access to them; (iii) can maintain dynamics, having processes of its own, independent of agents (e.g., evaporation, spontaneous growth or resources); (iv) is locally observable and accessible to agents, supporting agents' perception and action; and (v) can define/enforce rules for the MAS (e.g., mobility).

The above environment responsibilities are inter-related. For instance, the agents' perception of their neighborhood is related to the environment topology and its notion of location and vicinity. Macal and North (2014) identified the following common topologies: *soup*, where no spatial location is provided (a non-spatial model); *grid or lattice*, composed of cells–agents are located on them; *euclidean space*, composed of coordinates, which specifies agent location; *GIS*, composed of realistic patches of geospatial landscapes (the agent location can be a ZIP code or a geospatial coordinate); and *network*, which specifies static or dynamic links between nodes.

Interaction is a fundamental component in MAS. Often, the global system dynam-

ics emerges from the local behavior and interactions among its composing parts, characterizing a complex system. Interaction can be direct or indirect (BANDINI; MANZONI; VIZZARI, 2009). Direct interaction implies that an agent communicates directly (e.g., message exchanging). In order to communicate directly, agents must know each other and must acquaint with the communication model used (language, messages, protocol, etc.). Indirect interaction, in turn, uses some media interposed among the communication partners. Such media can be an artifact (e.g., a blackboard, a traffic light) or the environment itself (e.g., pheromone trails). The state of the media reifies the communication.

Interaction has a structure, which specifies relationships: what is or can be connected to what. The topology of the environment plays a key role in specifying this structure if inter-agent interactions are limited, for example, to the spatial neighborhood or to the agent acquaintance network. Interaction dynamics may change how, when, and which agents interact over time (MACAL; NORTH, 2014). Such dynamics is often governed by agents' deliberation mechanism.

The inference—the third step of the process for modeling and simulation in the ABMS paradigm—is done by simulating the execution of agent behaviors and interactions repeatedly over a timeline (MACAL; NORTH, 2014). Such simulation requires an execution specification, which must deal with the following aspects: (i) the temporal scale and extent of the simulation; (ii) dynamic processes scheduling; (iii) set-up of the model (its input data, how many entities must be created, and how they are initialized); and (iv) observation, specifying which data are collected for testing, understanding and analyzing the system, and how and when these data are collected (BANDINI; MANZONI; VIZZARI, 2009; GRIMM et al., 2010).

Explicit documentation for all these agent-based simulation details is key to promote models' understanding and replication. One step towards this direction was the Overview, Design concepts, and Details (ODD) protocol (GRIMM et al., 2006). The ODD protocol promotes a more rigorous formulation of models and provides a checklist for all the key elements of an agent-based simulation. These elements are grouped into three blocks, as shown in Table 2.1, which also presents the elements that should be documented for each block. The overview block intends to document the overall purpose and structure of the model so as to readers can get an idea of the model's focus, resolution, and complexity quickly. The design concepts block describes the general concepts from agent systems and complex adaptive systems underlying the design of the model. The last block, details, intends to present the details that were omitted in the overview block, in

Table 2.1: Elements of the ODD Protocol (GRIMM et al., 2010).

Category	**Element**
Overview	Purpose
	Entities, state variables, and scales
	Process overview and scheduling
Design concepts	Basic principles
	Emergence
	Adaptation
	Objectives
	Learning
	Prediction
	Sensing
	Interaction
	Stochasticity
	Collectiveness
	Observation
Details	Initialization
	Input data
	Submodels

particular, the submodels implementing the model's processes.

ODD is being widely used for the description of agent-based simulations (MÜLLER et al., 2014). In fact, the Journal of Artificial Societies and Social Simulation strongly encourage authors to document their models using ODD before publishing them (JASSS, 2017). However, ODD is completely based on textual descriptions. Although it is possible to include diagrams in an ODD document, there is neither standard notation nor ways to translate them to code.

2.1.2 Applications

A clear understanding of a model is fundamental to avoid either its misinterpretation or incorrect replication. ABMS is a step towards accuracy and precision in modeling, given that the close correspondence between the MAS model and the target system reduces the representational gap (EDMONDS, 2001).

Many existing phenomena can be modeled as agent-based systems. However, given the bottom-up nature of this paradigm, ABMS is particularly suitable for modeling complex systems and emergent phenomena. In these systems, dynamics is drawn from flexible and local interactions. Additionally, population sizes and environment structures can vary over time. These characteristics are not easily handled by conventional simulation paradigms, such as equation-based ones. Additionally, equation-based methods re-

quire prior knowledge of the aggregate phenomenon and its global patterns, which might be unavailable if the emergent phenomenon is unknown in advance (WILENSKY; RAND, 2015).

In equation-based models, entities are treated either as probability distributions or aggregated into variables. This assumption of population homogeneity is acceptable only when the population is sufficiently large so that the differences are averaged out, or an explicit treatment of heterogeneity does not lead to further gains. Unless appropriately used, these assumptions may lead to oversimplifications resulting in irrelevant simulation output (KLÜGL; BAZZAN, 2012). Consequently, the ABMS paradigm is suitable for systems that require heterogeneity in states and behaviors.

The ABMS paradigm is also suitable for systems where agents are adaptive towards their goals. In such systems, agents can be endued with learning or evolutionary capabilities to adapt to changes either in themselves or in the environment. Artificial intelligence techniques that provide such capabilities are well established and can be easily incorporated in simulations, allowing more realistic results (KLÜGL; BAZZAN, 2012).

Some trade-offs must be considered before adopting the ABMS paradigm. If the system has thousands or millions of agents, its simulation can be computationally intensive. The ABMS paradigm is focused on modeling at the micro level, and performance degradation is the price one pays for having a rich representation of agents. Equation-based models may be an alternative for performance. However, numerically solving complicated equation-based models may take as much computational time as agent-based models (WILENSKY; RAND, 2015).

The number of decisions that the model designer makes increases as more details must be incorporated into the model. Equation-based models are focused on describing the system at a macro (aggregate) level and rely on implicit assumptions about the micro level. In the ABMS paradigm, in turn, designers must know how the micro-level elements of the system operate, and thus they must decide how these elements will be specified as well.

The ABMS paradigm has been used to model and simulate systems in many application areas, such as agriculture, air traffic, ecology, economics, epidemiology, healthcare, market analysis, and social networks (MACAL; NORTH, 2014). According to Macal and North (2014), in these cases, ABMS was selected as the simulation paradigm because it can explicitly incorporate the complexity arising from individual behavior and interactions that exist in real-world scenarios.

2.2 Model-driven Development

Software engineering provides support for professional software development by means of techniques for specification, design, and evolution of systems. Software models are widely used in software engineering. In model-based development, models are used as guidance to source code implementation, playing a secondary role in the software development process. In Model-driven Development (MDD) (SCHMIDT, 2006), in contrast, models are *first-class citizens*, and the development is driven by modeling artifacts (ATKINSON; KÜHNE, 2003; STAHL et al., 2006a). In MDD, models are used to (semi-)automatically generate the source code of software systems and thus play a leading role in the development process. This (semi-)automatic generation of source code improves efficiency, productivity, reliability, reusability, and interoperability.

Previous work on the use of MDD in domains such as automotive manufacturing, mobile devices, and telecommunications, showed that productivity increases because the modeling effort is focused on *domain concerns* instead of programming statements (SPRINKLE et al., 2009). In the context of software development in industry, Tolvanen and Kelly (2016) showed that MDD approaches can increase productivity by a 5–10x factor.

Making domain abstractions available for modeling is fundamental in MDD. The goal is to allow building models that are simultaneously abstract and formal. Abstraction, in these models, does not mean vagueness, but compactness and reduction to the essence (STAHL et al., 2006a). The goals of MDD are the following:

- To increase development speed through automation of source code generation;
- To enhance software quality through formally-defined modeling languages and automated transformations;
- To avoid redundancy and manage technological changes: bugs can be fixed and cross-cutting concerns can be changed in just one place (i.e., the source code generator);
- To promote reusability: once modeling languages and transformations have been defined, they can be used in the sense of a software product line;
- To manage complexity through abstraction: modeling languages enable programming on a more abstract level.
- To offer a productive environment via the use of building blocks and best practices; and

- To meet interoperability and portability requirements by means of separating the specification of a particular functionality from its implementation on a specific platform;

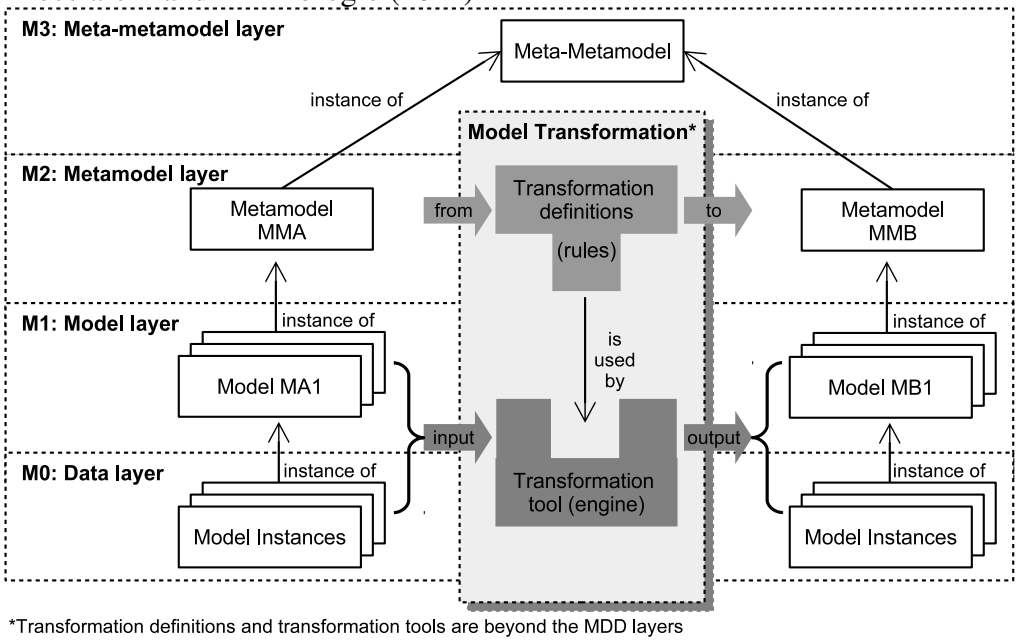
According to Stahl et al. (2006b), an MDD approach combines the following elements: (i) domain-specific languages; (ii) model transformations; and (iii) transformation engines and source code generators. The former allows expressing domain concepts effectively, while the latter two introduces automation. These elements are described as follows.

A Domain-specific Language (DSL) is a language designed for a particular domain, trading generality for expressiveness (STAHL et al., 2006a). The type system of a DSL formalizes the application structure, behavior, and requirements of that particular domain (SCHMIDT, 2006). Therefore, DSLs provide high-level, off-the-shelf, abstractions for the business-related concepts and process. For instance, a DSL for the simulation of dynamical systems (e.g., Simulink) would offer abstractions related to electric motors and power control. General-purpose languages, such as Java, can be used for simulating these elements, but they would demand additional effort and programming skills to build such abstractions from scratch. By reducing the amount of programming expertise needed, DSLs open up their application domain to a larger group of users (MERNIK; HEERING; SLOANE, 2005). The focus on a particular domain—in our case, ABMS—brings reading and learning efficiency and allow expressing *what*, instead of *how*, to compute (CONSEL et al., 2005; KOSAR et al., 2010). According to Tolvanen and Kelly (2016), DSLs can increase productivity by a 5x–10x factor. However, creating a DSL can be costly and difficult. Previous work on MDD showed that the more specific the DSL, the higher the chance of success (HUTCHINSON; ROUNCEFIELD; WHITTLE, 2011).

A common way of describing DSLs is metamodeling (SCHMIDT, 2006). A metamodel defines concepts and relationships among them, in addition to their key semantics and constraints within a domain. The construction of a metamodel follows from a metamodeling activity, on which a domain analysis is performed to identify the domain concepts that should be provided by a DSL and thus must be included in the metamodel (STREMBECK; ZDUN, 2009). Usually, a metamodel is constructed upon a meta-metamodel that provides elementary concepts to define metamodel elements, such as the ability to represent relationships between them. A metamodel is thus an instance of a meta-metamodel. A model of a system, in turn, is an instance of the domain metamodel. Meta-metamodels, metamodels, and models represent different abstraction layers of an

MDD approach. These layers are organized by their abstraction level, as illustrated in Figure 2.2. The meta-metamodel is at the top (M3) layer, followed by the metamodel (M2) and model (M1) layers. The bottom layer (M0) contains model instances, which represent particular systems.

Figure 2.2: MDD Abstraction Layers and Model Transformation Elements. Adapted from Bocciarelli and D'Ambrogio (2014)



While the metamodel describes the domain concepts available for building models (in other words, the language abstract syntax), the notation used to render these concepts is specified by the language concrete syntax (ATKINSON; KÜHNE, 2003). An effective concrete syntax reduces the abstraction gap by providing building blocks that expressively represent domain concepts, leading to a more productive environment. The precise meaning of the symbols used in the concrete syntax is formalized by the semantics of the language. The semantics of a DSL must be either well-documented or intuitively clear, adopting concepts from the problem space so that a user having domain knowledge will recognize the domain elements (STAHL et al., 2006a).

The goal of the second element of an MDD approach, model transformations, is to convert models. Model transformations are automated by means of transformation engines and source code generators, and those constitute the third element of an MDD approach. With a transformation engine, a source model can be converted to a target model or to code. The former conversion is called model-to-model (*m2m*) transformation, and the latter is model-to-code transformation (*m2c*). Transformation rules are specified at the metamodel level to map elements of distinct metamodels or to map elements to code state-

ments, as illustrated in Figure 2.2. For example, transformation rules can map elements of class diagrams from the UML to statements of the Java programming language. Model-to-code transformations are the basis of source code generators that, in turn, automate the creation of low-level software artifacts (e.g., lines of code). Manually creating such artifacts is an effort-consuming task and requires technical expertise. Moreover, pieces of code for recurrent structures and concepts are often repeatedly implemented. By putting these pieces of code into code generators, an MDD approach increases productivity in software development.

2.3 Related Work

Remarkable work has been developed in the MAS community for modeling or building agent systems. Gómez-Sanz and Fuentes-Fernández (2015) reviewed a selection of Agent-Oriented Software Engineering (AOSE) methodologies in order to account how well the AOSE community is meeting the software lifecycle used in the industry. Their study considered consolidated methodologies that have received attention from the community in the last years (GAIA, MESSAGE, PASSI, Tropos, Prometheus, INGENIAS, ADELFE, and MaSE) as well as new methodologies (ASEME, GORMAS, OperA, DSML4MAS, and ForMAAD). There are also modeling languages tailor-made for MASs: Agent-DSL, MAS-ML, SDLMAS, DSML4MAS, Agent-UML, AM, and SEA_ML. Additionally, many agent-based platforms are available for creating MASs. In a recent work, Kravari and Bassiliades (2015) surveyed twenty-four platforms that are used by the MAS community. Considering such magnitude in the number of platforms, we focus on the ones providing support for simulation.

MDD alternatives were also conceived to build MASs. Kardas (2013) reviewed a selection of MDD approaches for MAS and classified them into three categories: complete MDD approaches with multiple metamodels at distinct abstraction levels and model transformations (Cougaar MDA, PIM4Agents, and MDD4SW Agents); partial MDD approaches with single metamodels and either model-to-model or model-to-code transformations (Malaca, CAFnE, TAOM4e, PIM4SOA, AMDA); and extensions of existing MAS methodologies to support MDD (INGENIAS Development Toolkit (IDK), Model-driven Tropos, and Model-driven ADELFE).

From the point of view of the ABMS paradigm, previously mentioned approaches share a common drawback: they are focused exclusively on the MAS elements. In that

sense, they are approaches for constructing general purpose MAS. However, built-in agent abilities, such as learning and adaptation, are not provided. Except for some extensions for these work (detailed later), they do not cover simulation aspects such as temporal extent, exogenous processes, initialization, observation, and repetition. Even having total or partial MDD support, the focus on general purpose MAS prevents these approaches from providing high-level abstractions for specifying such execution aspects. Moreover, Kardas (2013) noticed that in most of these MDD approaches, support for code generation is limited. Usually, code is generated only at the template level, and a significant amount of code needs to be manually completed. Therefore, feasibility and effectiveness of these MDD approaches are limited because the amount and quality of the automatically generated MAS components appear to be insufficient.

Considering that this thesis focuses on agent-based simulations, the review is narrowed to methodologies and platforms able to deal with simulation aspects. Given that this thesis proposes an MDD approach agent-based simulation development, alternatives that refer to at least one MDD concern are considered: metamodels, or domain-specific languages, or model transformations. The reviewed approaches are classified into two categories: metamodels (which also includes methodologies and processes) and platforms. In the following sections, we briefly describe these approaches and discuss their limitations.

2.3.1 Metamodels

The model-driven approach of Iba, Matsuzawa and Aoyama (2004) is based on a metamodel that covers general aspects of agent-based simulations, such as agents, behaviors, relations, goods, and information. The approach relies on the UML for creating simulation models on the basis of their proposed metamodel. While behaviors are specified with UML activity, communication, and statechart diagrams, all other elements are specified with UML class diagrams.

Garro, Parisi and Russo (2013) proposed MDA4ABMS, a process based on the model-driven architecture for specifying simulation models. A light, task-based metamodel is provided for modeling structural and behavioral aspects of agents and the environment. As in previously mentioned approaches, MDA4ABMS relies on UML for modeling these aspects. Moreover, code generation is semi-automated. Guidelines are provided for manually transforming some elements of diagrams into code artifacts.

As reported by Bauer and Odell (2005), UML does not provide expressive constructs to specify high-level aspects of agent-based simulations. Aspects such as adaptation and learning need to be specified from scratch, as well as simulation aspects. Therefore, effectiveness and productivity are compromised in these UML-based related work.

Alternatively, there are MDD approaches for agent-based simulation development that do not rely on UML for modeling. These approaches range from brand new metamodels with modeling languages and model transformations to extensions of existing agent methodologies.

The AMASON metamodel (KLÜGL; DAVIDSSON, 2013) was proposed to capture the basic structure and dynamics of agent-based simulations. According to its authors, AMASON is focused not on providing a highly elaborated metamodel with specific suggestions to all possibly occurring concepts, but on a metamodel that fits a wide variety of models. Consequently, it covers only very basic structures and processes. Models are specified as tuples and sets of bodies, minds, and regions (environment), and their dynamics is specified by means of predicates, which define state transitions. Neither model transformations nor code generation is provided.

The MAIA metamodel (GHORBANI et al., 2013) captures social concepts such as norms and roles. It is based on institution analysis and therefore is mainly focused on agent-based social simulations. In (GHORBANI et al., 2014), semi-automatic code-generation was proposed in the form of a guideline for transforming a MAIA model into an executable simulation. Human intervention is required to transform aspects such as the simulation setup. Social concepts are also considered by Coutinho et al. (2009), which proposed an organizational metamodel by merging existing organizational models. Such a merged metamodel promotes organizational interoperability among organizational models. However, simulation designers need to specify the organizational model from scratch, given that the merged metamodel does not capture models recurrently used in simulations.

In the IODA methodology for agent simulations (KUBERA; MATHIEU; PICAULT, 2008), a modeling language was proposed for specifying the simulation dynamics as interactions among agents. An interaction is considered a condition/action rule involving a source and a target agent. The concrete syntax of the modeling language is characterized by an interaction matrix, which indicates which interactions an agent may perform or undergo, and with which other agents. However, there are no available coarse-grained building blocks and thus complex interactions must be specified from scratch.

The use of the INGENIAS (PAVÓN; GÓMEZ-SANZ, 2003) modeling language to

specify agent-based simulation models was proposed by Fuentes-Fernández et al. (2010). The INGENIAS language provides elements for modeling agent-related aspects such as agents, roles, goals, tasks, events, interactions, and societies. Similarly, Sansores and Pavón (2005) adopted the INGENIAS language for modeling and proposed model transformations to generate code for agent-based simulation platforms. In both work, further customizations in the INGENIAS language were required to support the modeling of simulation aspects, such as the temporal and spatial extent (SANSORES; PAVÓN; GÓMEZ-SANZ, 2005) and the scheduling and management of agents' life-cycle (GÓMEZ-SANZ; FERNÁNDEZ; ARROYO, 2010). In spite of providing improved representation for agent-related aspects in comparison to UML, the INGENIAS language does not provide expressive elements for recurrent aspects such as adaptation and learning. Additionally, as reported by Kardas (2013), the INGENIAS development kit has issues with respect to code generation—code is generated only at the template level.

The presented approaches have limited support for modeling sophisticated aspects of agent-based simulations and for generating code from them. Table 2.2 presents a comparison of the described approaches. Some approaches adopt UML diagrams as the means of specifying the simulation structure and dynamics, while others adopt textual and tabular descriptions. These approaches provide neither built-in simulated environment nor agent capabilities.

Nevertheless, it is noteworthy to discuss how these approaches have been evalu-

Table 2.2: Comparison of Metamodels for Agent-based Simulation

Approach	**Means of Specification***	**Built-in simulated environments**	**Built-in agent capabilities**
Iba, Matsuzawa and Aoyama (2004)	UML class, activity, communication, and statechart diagrams	-	-
MDA4ABMS	Textual description Tables UML activity diagram	-	-
AMASON	Tuples and sets Predicates for state transitions	-	-
MAIA	Tables and forms	-	-
IODA	Interaction matrix	-	-
INGENIAS	INGENIAS diagrams	-	-

*of agents, interactions, and environment (structure and dynamics)

ated. As previously mentioned, MDD is effective for increasing productivity in software development if the development effort is focused on domain concerns instead of programming statements. Such a gain in productivity would not be possible if the effort to create models using the provided metamodels and modeling languages were greater than using other ways (e.g., programming languages). Therefore, in order to show that MDD approaches are *effective* and thus improves productivity, one must go beyond showing that they are *feasible*. With respect to related work on MDD targeted to agent-based simulations, we observed that they limit themselves to showing examples and case studies that demonstrate the *feasibility* of their MDD approaches—the ability to model and, in some cases, to generate code. No *effectiveness* evaluation is presented. Therefore, despite being able to generate code from models, there is no evidence that the effort required to create these models is less than developing simulations directly on the target simulation environment.

2.3.2 Platforms for Agent-based Simulation

We next present and discuss agent-based simulation platforms. The discussion is focused on the availability of features and building blocks for aspects recurrently used in simulations. We refer the reader to Sichman (2015), and Kravari and Bassiliades (2015) for a comparison of these platforms regarding other, more general aspects. We limit the discussion to the platforms enumerated by Klügl and Bazzan (2012): Swarm, Repast, MASON, and NetLogo. Kravari and Bassiliades (2015) reported that these platforms belong to the group of the most promising ones. We also considered two additional platforms: GAMA and AMP. Table 2.3 presents basic characteristics of these platforms, sorted by the latest release date. It can be seen that these platforms are distributed under public/academic license and are open source. Although MASON, Swarm, and AMP are yet considered in recent surveys, it has been a while since their latest releases.

GAMA and NetLogo provide their own graphical toolkit and programming languages for developing agent-based simulations (TAILLANDIER et al., 2010; WILENSKY, 1999). In their graphical environment, designers can specify a few details of the simulation, and control its execution by means of user interface components. Each platform provides its own programming language, in which the dynamics of agents, interactions, and simulated environment must be programmed. Therefore, the designer should have programming expertise and have had learned these programming languages to de-

Table 2.3: Basic Characteristics of Agent-based Simulation Platforms. Adapted from (KRAVARI; BASSILIADES, 2015)

Platform	Latest release	License	Open source
GAMA	September 11, 2018	public/academic	√
NetLogo	June 14, 2018	public/academic	√
Repast	October 27, 2017	public/academic	√
SeSAm	February 06, 2017	public/academic	√
MASON	June 19, 2015	public/academic	√
Swarm	August 01, 2013	public/academic	√
AMP	May 27, 2012	public/academic	√

velop simulations. Concerning the simulated environment, NetLogo provides only a grid topology, with methods for importing environmental data from Geographic Information System (GIS) files. Any other topologies (e.g., a graph) need to be specified programmatically. GAMA provides both grid, graph, and Cartesian space topologies. Additionally, GAMA provides support for loading the simulated environment directly from GIS files. With respect to built-in agent capabilities, NetLogo provides code libraries for features such as fuzzy logic and linear programming. GAMA provides built-in abilities for specifying agent reflexes, tasks, and states—these abilities are composed of sets of actions executed automatically by the agent. No library is provided either by NetLogo or GAMA for agent capabilities such as learning or adaptation.

Repast provides a graphical toolkit with editors to develop agent-based simulations (NORTH; COLLIER; VOS, 2006). In Repast, a simulation is developed either in plain Java code or in the ReLogo programming language. Therefore, both the structure and dynamics of agents, interactions, and the simulated environment are specified as either Java or ReLogo source code. These elements are conceptualized as classes, methods, attributes, and relationships—from Object Oriented Programming (OOP). Recently, statecharts were provided to conceptualize agents' dynamics (OZIK et al., 2015). Within a statechart, the designer can specify agent states and transitions between them. However, elaborated agent dynamics still needs to be specified using either ReLogo or Java code within on enter/on exit events associated with states or transitions. Repast provides a set of OOP classes for recurrent environment topologies beyond grids, which could potentially ease the development of simulations in a great variety of domains. Concerning abstractions for agent capabilities, Repast provides built-in support for specifying neural networks and genetic algorithms, all of them by means of Java source code libraries.

MASON and Swarm are programming libraries for developing agent-based simulations (LUKE et al., 2005; MINAR et al., 1996). While MASON is a Java library,

Swarm is distributed either as a Java or an Objective-C library. Both provide a set of classes to develop agent-based simulations and to visualize the simulation execution. In these platforms, agents, interactions, and the simulated environment, as well as the simulation execution, need to be source coded. The set of classes provided by MASON includes classes for recurrent environment topologies, for rigid body dynamics, and for agents' processes scheduling. Swarm provides only a grid environment topology. With respect to agent capabilities, MASON provides classes for evolutionary computing concepts. That way, agents can be endued with features and operations to produce adaptive behaviors by means of genetic evolution. In contrast, no built-in feature is provided by Swarm for agent capabilities.

In SeSAm and AMP, agent-based simulations are developed by visual programming (KLÜGL; HERRLER; FEHLER, 2006; PARKER, 2010). Agents, interactions, and the simulated environment are assembled by means of a component tree. In SeSAm, the dynamics of these elements are specified with activity diagrams that resemble the UML activity diagram. In turn, in AMP, dynamics is specified as chained action nodes in the component tree. With respect to the simulated environment, while SeSAm provides only a grid topology, AMP provides grid, continuous space, and network environments. Both platforms do not provide built-in agent abilities, which means that abilities such as learning must be developed from scratch.

Table 2.4 presents a comparison of the discussed simulation platforms. It can be seen that these platforms have limited built-in support for specifying sophisticate aspects of agent-based simulations. Although it is possible to create simulations that consider agent abilities such as learning, flow control, and disease models, much must be developed from scratch. Moreover, developers must have expertise in programming logic to develop those abilities in either textual or visual programming languages. Finally, although simulation aspects (e.g., exogenous processes, simulation setup, and observation) are supported by these simulation platforms, programming expertise is usually required to specify them.

2.4 Final Remarks

The ABMS paradigm is particularly suitable for modeling emergent phenomena and complex systems composed of interacting heterogeneous agents with adaptive behaviors. Despite the potential of the ABMS paradigm, there is a lack of methodological

Table 2.4: Comparison of Platforms for Agent-based Simulation

Platform	Means of Specification*	Built-in simulated environments	Built-in agent capabilities
GAMA	GAMA code	Grid Graph Cartesian space	Agent reflexes, tasks, and states
NetLogo	NetLogo code	Grid	Fuzzy logic Linear programming (NetLogo code libraries)
Repast	ReLogo or Java code Statechart for agents' dynamics	Grid Continuous space Network (all by means of OOP classes)	Neural networks Genetic algorithms (Java code libraries)
MASON	Java code	Grid Continuous space Network (all by means of OOP classes)	Rigid body dynamics Agent process scheduling (Java code libraries)
Swarm	Java or Objective-C code	Grid	-
SeSAm	Visual component tree Activity diagram	Grid	-
AMP	Visual component tree Chained action nodes	Grid Continuous space Network	-

*of agents, interactions, and environment (structure and dynamics)

support. That includes all steps from a precise formulation of the simulation objective to automatic generation of model documentation (KLÜGL; BAZZAN, 2012). Additionally, researchers have already stressed the importance of building blocks and tools at a higher abstraction level (PARUNAK; SAVIT; RIOLO, 1998; HAMILL, 2010; KLÜGL; BAZZAN, 2012; WELLMAN, 2016). One step towards this direction is the ODD protocol. However, it is completely based on textual descriptions and hence provides neither a standard notation nor code generation.

As stated by Parunak, Savit and Riolo (1998), a widespread realization of the advantages of ABMS will depend on the availability of several intuitive, drag-and-drop tools for building and analyzing agent-based models easily. The popularity and maturity of equation-based modeling are explained by the availability of such tools. Most alternatives for agent systems are at lower levels of solution maturity, and the way towards consolidation of alternatives must include languages that support solutions to recurrent problems (WEYNS et al., 2015). This thesis vision goes towards this direction. With an approach that provides off-the-shelf concepts recurrently used in simulation (e.g., agent

capabilities such as learning, or topologies for the simulated environment), the adoption of ABMS as a simulation paradigm could possibly increase. Considering that MDD aims to find domain-specific abstractions and make them available for modeling, it is a candidate approach for raising the abstraction level in agent-based simulation development.

3 MODEL-DRIVEN AGENT-BASED SIMULATION DEVELOPMENT

In this thesis, we propose MDD4ABMS, a model-driven approach for agent-based simulation development. As described in the background chapter, making domain abstractions available for modeling is key to the success of any MDD approach. Therefore, the core of our MDD4ABMS approach is a metamodel that captures the aspects that compose agent-based simulations (agents, interactions and the simulated environment), in addition to the simulation execution specification. From such a metamodel, ready-to-run simulation code can be automatically generated via model-to-code transformations.

This chapter describes how the MDD4ABMS metamodel was conceived following a generative, bottom-up approach, and how it was extended via a domain analysis method to capture concepts from particular application domains. The metamodel is constructed upon the Ecore meta-metamodel[1], and it is specified with UML. This chapter starts with an overview of the elements that compose the MDD4ABMS approach and how they interact. Then, the core metamodel is explained. Finally, we describe two extensions to the core metamodel that capture concepts from the adaptive traffic signal control and spread of disease application domains.

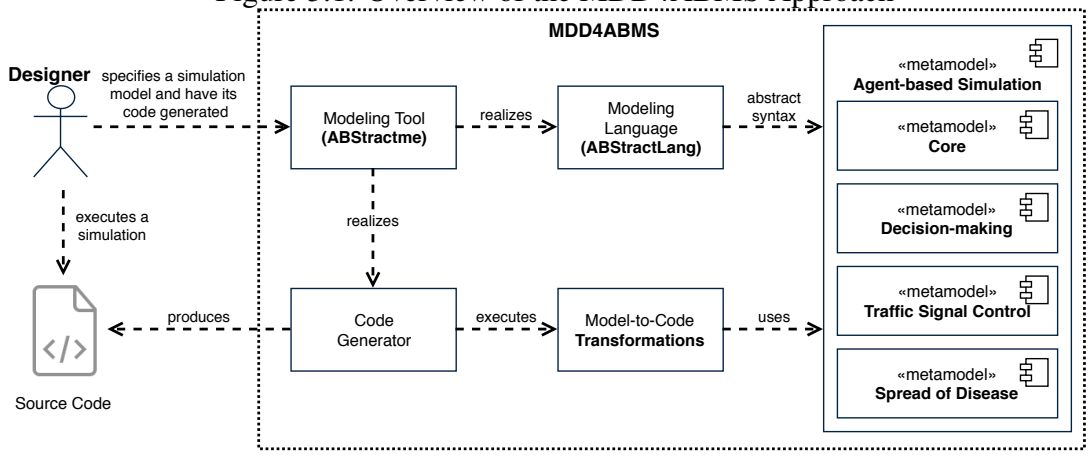
3.1 MDD4ABMS Overview

The main elements of our MDD4ABMS approach are (i) a metamodel for agent-based simulations, (ii) the ABStractLang modeling language, and (iii) model-to-code transformations. MDD4ABMS also includes ABStractme, a modeling tool through which designers can create simulation models and have their source code generated. Figure 3.1 shows an overview of our MDD4AMBS approach and its elements, which are briefly described next.

The agent-based simulation metamodel abstracts aspects recurrently used in simulations. Basic aspects, such as the simulated environment, entities, and agents, are grouped into a core metamodel. Other aspects, such as decision capabilities and concepts that are domain specific, are incorporated into specific metamodels. Together, these specialized metamodels compose the agent-based simulation metamodel. This metamodel can be further extended to capture aspects from other application domains. Indeed, the simulation metamodel proposed in this thesis has been already extended with the fol-

[1] <http://www.eclipse.org/modeling/emf/>

Figure 3.1: Overview of the MDD4ABMS Approach



lowing concepts from two application domains: decision making and flow control from the traffic signal control domain, and compartmental disease models from the spread of disease domain.

Specifying a simulation by manually instantiating metamodel elements would be an effort consuming task. To ease the specification of simulations MDD4ABMS provides ABStractLang, a modeling language that provides building blocks to reduce the abstraction gap and to represent domain concepts expressively. The agent-based simulation metamodel defines the abstract syntax of the ABStractLang language.

In the model-to-code transformations element, transformation rules are specified on the basis of the metamodel. These transformation rules are executed by the code generator element, which receives as input a simulation model, executes the model-to-code transformations, and produces as output the NetLogo source code that implements the model. Finally, the ABStractme modeling tool element provides an integrated environment on which the designer can specify agent-based simulations using the ABStractLang modeling language and have its source code generated.

In the following sections, we detail the core and the specialized metamodels and how they were constructed. The modeling language, model-to-code transformations, and modeling tool elements are described later, in Chapter 4.

3.2 Core Metamodel

Any source of explicit or implicit domain knowledge can be used as input for identifying domain abstractions and build a metamodel, such as technical documents, knowledge from domain experts, and existing code (MERNIK; HEERING; SLOANE, 2005).

In this thesis, we built the metamodel considering existing agent-based simulations. This generative, bottom-up approach was used to incorporate into the metamodel only those elements that are, indeed, recurrently used in agent-based simulations.

The core metamodel abstracts basic, recurrent concepts found in existing agent-based simulations. To build this metamodel, we considered a set of 50 existing agent-based simulations available at the CoMSES Net Computational Model Library (ROLLINS et al., 2014; CoMSES Net, 2018). This set was composed of the 45 last recently published simulations by the time of the analysis (Dec, 2015), and the 5 top-downloaded ones (MURPHY, 2015). Simulations with neither associated publication or available source code were removed, resulting in a set of 17 simulations whose domains are social simulation, land use, and epidemic dissemination. We included an additional existing simulation that was not available at the CoMSES model library: the Haiti Earthquake simulation (CROOKS; WISE, 2013), which investigates how people react to the distribution of food, and how rumors relating to their availability propagate through the population. Finally, the Sugarscape simulation (EPSTEIN; AXTELL, 1996; LI; WILENSKY, 2009) was included due to its potential for being used with educational purposes. Therefore, a set of 19 simulations were considered to build the core metamodel.

Each simulation was investigated to elicit the elements that describe core aspects. The ODD protocol was used to guide such elicitation, in particular, its *overview* and *details* blocks in which structural simulation elements are documented, and the *observation* element of the *design details* block in which the data collected from the simulation is specified. We identified that these simulations use parameters as input, and some of them rely on external files to create or initialize their environments, such as GIS files with elevation or demographic data. Such simulation types often demand high coding effort, since developers typically need to program data input routines in the simulation platform. In the Haiti Earthquake simulation, for example, external GIS files are used for the specification of the environment, containing information about the population density, relative level of devastation, existing transportation network, and location of aid centers. After investigating all the simulations, we identified the following recurrent elements: an *environment*, in which all *situated entities* are located; simulation *parameters* (e.g., rates, amounts, etc.); and *entities and agents* with *attributes* and *relationships*. These simulation elements have *structural* and *execution* (setup) aspects, as well as outputs. Next sections describe how these elements are abstracted in the metamodel.

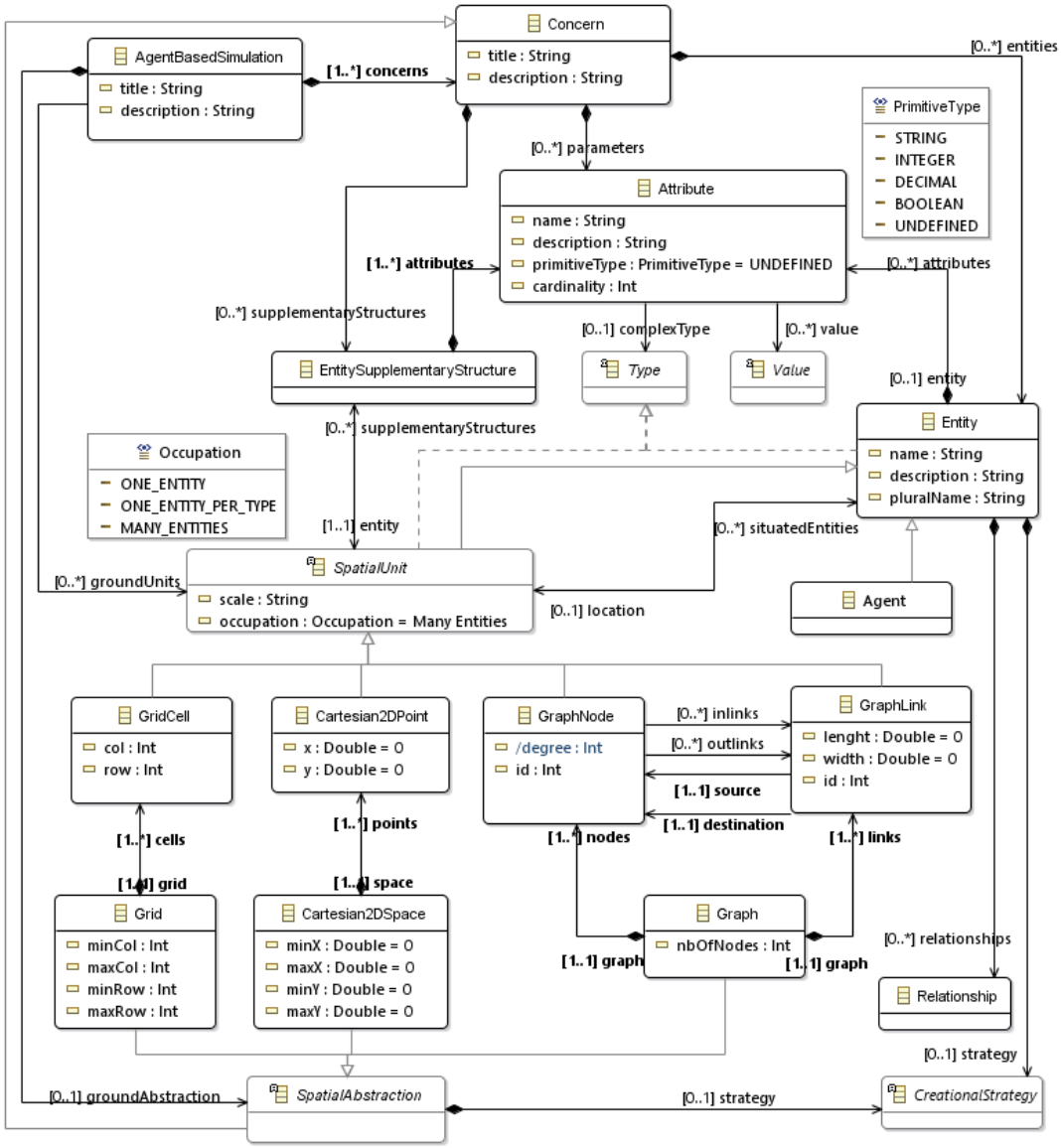
3.2.1 Basic Simulation Aspects

Figure 3.2 shows a partial view of the core metamodel in which the basic simulation elements identified are abstracted. The main element of the metamodel is the `AgentBasedSimulation`, which aggregates all other simulation elements. To specify the environment in which all other situated entities (`Entity`) are located, the metamodel introduces the notion of spatial abstraction. `SpatialAbstraction` is the *ground* of the simulation, discretized in spatial units (as proposed in the ODD protocol). A `SpatialUnit` is a particular type of entity in which other objects of the environment can be situated. Based on the investigated simulations, three types of spatial abstractions are provided, each of them with its particular spatial unit: (i) `Grid` composed of cells (`GridCell`), (ii) `Cartesian2DSpace` composed of points (`Cartesian2DPoint`), and (iii) `Graph` composed of links and edges (`GraphLink` and `GraphNode`, respectively). A spatial unit also specifies an `Occupation` policy, which affects the mobility of agents. Three occupation policies are provided: many entities or agents can simultaneously occupy the same spatial unit; distinct entities can occupy the same spatial unit given that they are of distinct types (e.g., a human and a pet agent); or, a more restrictive policy in which only one entity can occupy the spatial unit per timestep.

Spatial units have standard properties, i.e., a `GridCell` has a `col` and `lin` values that defines its location (coordinates) within the `Grid` spatial abstraction. However, additional properties may be required depending on the simulation. To support spatial unit customization, the `EntitySupplementaryStructure` was incorporated into the metamodel to allow supplementing the spatial unit with additional attributes, which are then embedded into the spatial unit entity.

In the considered simulations, model parameters and entities are often implemented without a clear separation of the concepts associated with different aspects of the simulation. Such a situation can lead to scattered descriptions across distinct portions of the simulation model or code, which can impair comprehension and future maintenance. To avoid such scattering, our metamodel groups the specification of parameters and entities by `Concern`. Each concern has a title and a description, and can be composed of parameters, entities, and supplementary structures. This concern-driven specification is helpful when modeling simulations involving experts from different application areas or sub-areas. It also promotes scalability and manageability of models, given that concerns split models into smaller, manageable parts, as recommended in the MDD litera-

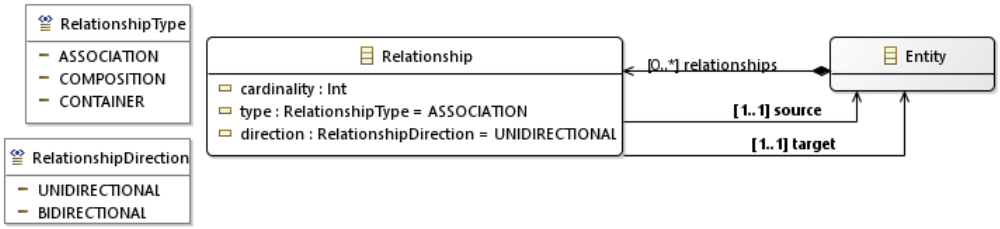
Figure 3.2: Core Metamodel: Environment, Entities, and Agents



ture (VOELTER et al., 2013).

An `Entity` is a relevant object that exists in a simulation. The structural specification of an entity includes its name and attributes. An `Agent` is a particular kind of entity that may have *capabilities*. Concerns receive parameters usually used as input to a simulation, which are concern attributes (e.g., the number of buildings). Entity and concern attributes correspond to the same model element, namely `Attribute`. Such attributes are described in terms of name, type, and cardinality, i.e., whether this value is a collection and of how many elements. The attribute type is either a `PrimitiveType` or a complex type represented by the `Type` element. An attribute is associated with a value when simulation runs. The idea of entities as elements with attributes is in fact part of almost all

Figure 3.3: Core Metamodel: Relationships



agent-based metamodels, such as those presented by Bernon et al. (2004). However, given that we follow a bottom-up approach, existing metamodels were not used as starting point to avoid bias that can lead to the introduction of unnecessary or overly complex concepts.

Relationships can be specified between entities. Figure 3.3 shows the relationship types abstracted into the core metamodel.[2] A `Composition` relationship indicates that the source entity is composed of other entities, representing a whole-part relationship. The whole is responsible for creating the parts, and the whole does not exist without its parts (e.g., a building is composed of its floors). A `Container` relationship indicates that the source entity may contain others, e.g., a building may contain people. Finally, an `Association` indicates relationships with other semantics between two entities (e.g., friendship between people). A relationship has a cardinality, which specifies how many target entities may be related to the source entity during the execution of a simulation. A relationship also has a direction, which specifies the *awareness* of the relationship. If `UNIDIRECTIONAL`, only the source entity is aware of the relationship. If `BIDIRECTIONAL`, both source and target entities are aware of the relationship.

3.2.2 Creation of Entities and Initialization & Update of Attributes

The elements previously presented are related to the structural aspects of an agent-based simulation. Additionally, the metamodel has elements to describe the execution specification: how values of attributes are initialized and updated, and how entities and agents are created. A `Source` element, shown in Figure 3.4, abstracts the provision of values, hiding low-level details of how these values are provided. Sources are used both to initialize or update attributes, and to provide values to any other simulation element that demand them (e.g., a learning technique demand values for its parameters, such as the

[2] In this and in all other diagrams in the following, the gray color is used to highlight the metamodel elements that were either already described or will be detailed later.

Figure 3.4: Core Metamodel: Sources

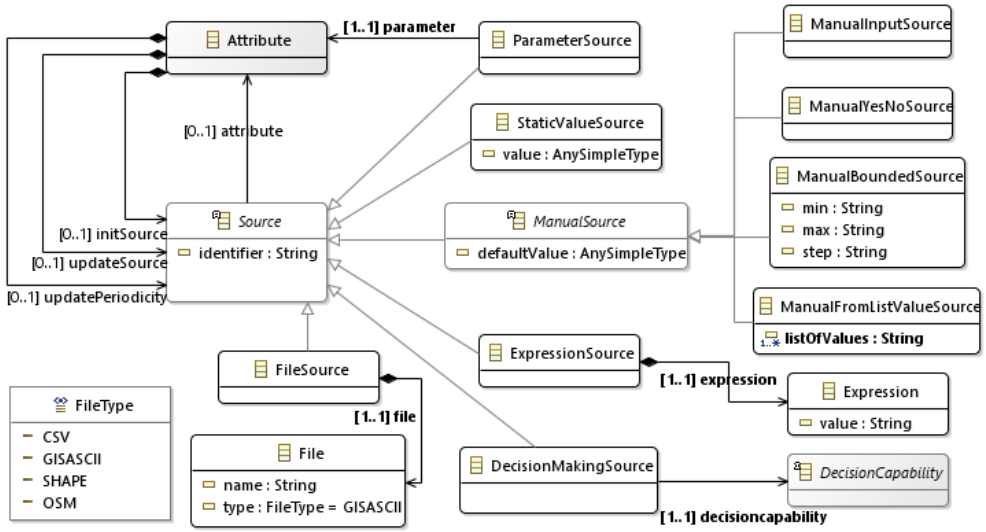


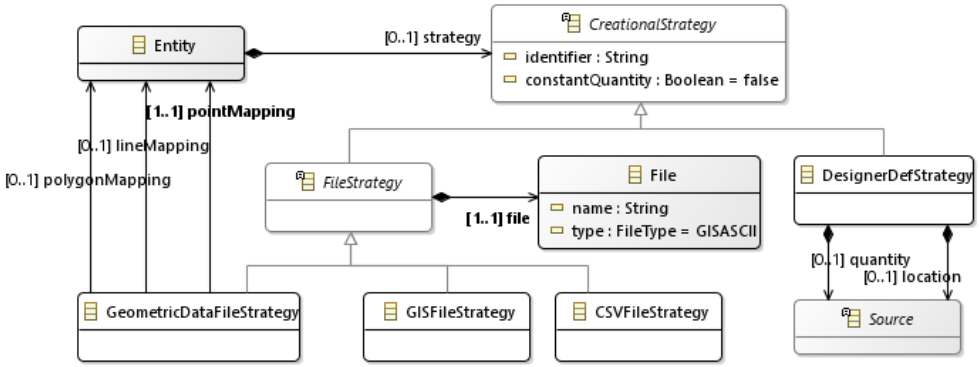
Table 3.1: Core Metamodel: Types of Sources

Type	Identifier	Description
`StaticValueSource`	**V**	The designer informs the value directly in the specification.
`ManualSource`		The value is provided at runtime, by means of some input component. Supported manual sources are the following:
`ManualInputSource`	**MI**	Manual input that accepts string or numeric values.
`ManualYesNoSource`	**MYN**	Manual yes/no input, that may be used for boolean values.
`ManualBounded Source`	**MB**	Manual input of numeric values. This source creates an input component that allows numeric values bounded by a *min–max* interval and that are multiples of a *step* value.
`ManualFromList ValueSource`	**MLV**	Manual input that provide a list of allowed values.
`ExpressionSource`	**E**	An `Expression` element specifies operations over values or parameters to compute the resulting value (e.g., `math` operations).
`ParameterSource`	**P**	The provided value is the value of a previously defined parameter.
`FileSource`		Values are read from files. Supported file formats are the following:
`CSV`	**FCSV**	The value is read from a Comma Separated Values (CSV) file. The attribute of the n^{th} created entity is set according to the specified fields of the n^{th} line in the file.
`GISASCII`	**FGIS**	The value is read from an ASCII GIS file, a tabular data file in which each value is associated with a point in the space. The attribute of the entity located at (x,y) has its value set according to the element at (x,y) in in the file.
`SHAPE`	**FSHP**	The value is read from a shape file, which contains geospatial vector data structured as points, lines or polygons in the space. The attribute is set according to the specified attribute from the corresponding shape of an entity in the file.
`OSM`	**FOSM**	The value is read from an Open Stree Map (OSM) file, which is a structured data file that represents traffic networks. The attribute is set according to the specified attribute from the corresponding traffic link or node in the file.
`DecisionMakingSource`	**DM**	The value is provided by a decision capability (described later).

learning rate). Table 3.1 shows the types of sources provided (the *Identifier* column shows how source types are referred to in the ABStractLang modeling language, described afterwards). Only `StaticValueSource`, `ManualSource`, and `ExpressionSource` are accepted to initialize attributes associated with concerns (i.e. parameters). To initialize and update entity attributes, in turn, any of the provided sources are accepted. These association constraints between attributes and sources are part of our metamodel semantics.

Creational strategies are incorporated into the metamodel to abstract the setup of the simulation, in particular how entities and agents are created and located in the spatial abstraction. These are represented by the `CreationalStrategy` element, as shown in Figure 3.5. Two strategies are supported: `DesignerDefStrategy` and `FileStrategy`. These are detailed in Table 3.2. Although file-based strategies are based on the same file types, creational strategies and sources give distinct semantics for file contents.

Figure 3.5: Core Metamodel: Creational Strategies



3.2.3 Data Collection

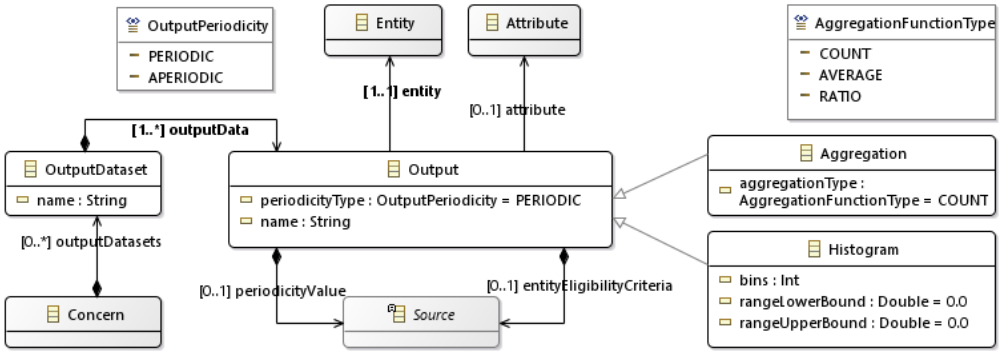
With respect to data collected for testing, understanding, and analyzing the simulation, the considered simulations usually output these data to plots or files, either periodically or by the end of the simulation. Figure 3.6 shows the metamodel elements that capture the specification of which data is collected, and how and when they are collected. This specification is supported by the `OutputDataset` and `Output` elements. The latter abstracts a particular value collected from an entity, while the former groups related values that are either saved in the same file (as columns) or shown in the same plot (as lines or bars). Each output has its own periodicity, which can be either `PERIODIC` (the value collected at every step of the simulation) or `APERIODIC` (the value is collected by the end of the simulation execution). Two output types are provided. In the `Aggregation` output,

Table 3.2: Core Metamodel: Types of Creational Strategies

Type	Identifier	Description
`DesignerDefStrategy`	**Designer Defined**	The designer is in charge of specifying the number of entities (*quantity*) that are created and their *locations*. The specification is done via *sources* to abstract the provision of such values. Only the following *sources* are accepted: for *quantity*: Static Value (**V**), Manual (only **MI**,**MB**, or **MLV**), Parameter (**P**), and Expression (**E**). In all these cases, the resulting value of the source must be an integer value. for *location*: Static Value (**V**), Manual (only **MI** or **MLV**), or Expression (**E**). The resulting value of the source must be the absolute position of the entity on the environment (e.g., (x,y) coordinate). For an Expression source, the result value can also be the name of another entity specified in the model. In such a case, the newly created entity will be located at the same position of one existing entity whose name was provided.
`FileStrategy`		Represents file-based strategies, on which the number and location of entities are provided as files. The following file strategies are supported:
`CSVFileStrategy`	**CSV File**	One entity is created for each line of the CSV file, and the comma separated fields are available for use in the initialization of its attributes (e.g., in expressions).
`GISFileStrategy`	**GIS File**	One entity is created for each data value, and its location is the spatial unit at that coordinate.
`GeometricDataFile Strategy`	**Shape File**	A composite entity and its contained entities (specified using *composition* relationships) are created according to the mapping of points, lines, and polygons to their corresponding entity types.

the value is collected via an aggregation function (either `COUNT`, `AVERAGE`, or `RATIO`). In the `Histogram` output, the value feeds a histogram function. If the output is `AVERAGE` aggregation function or `Histogram`, it must be related to the entity attribute that provides the value.

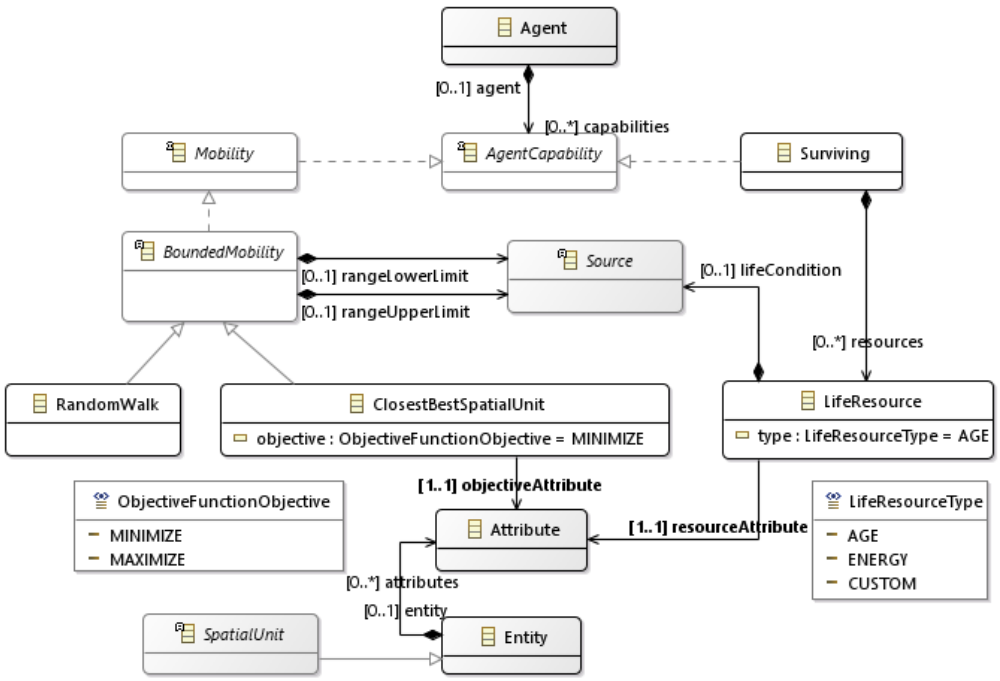
Figure 3.6: Core Metamodel: Simulation Outputs



3.2.4 Basic Agent Capabilities

As mentioned earlier, an agent is a particular kind of entity that may have capabilities. Figure 3.7 shows the metamodel elements of basic agent capabilities identified in the considered simulations. The `AgentCapability` abstracts the notion of agent capability. Two basic capabilities are provided: `Mobility` and `Surviving`.

Figure 3.7: Core Metamodel: Mobility and Surviving Agent Capabilities



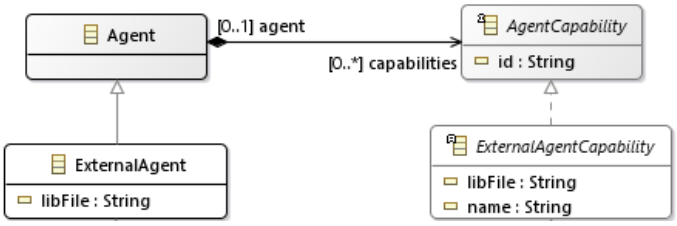
A `Mobility` element abstracts agents' ability to move from one spatial unit to another within the spatial abstraction (the simulated environment). The `BoundedMobility` element abstracts that kind of mobility on which the range is limited by either an upper limit, a lower limit, or both. Two subtypes of bounded mobility are provided: `RandomWalk` and `ClosestBestSpatialUnit`. While agents endued with a `RandomWalk` mobility move to a random spatial unit within the range limits, agents endued with a `ClosestBestSpatialUnit` mobility move to the closest spatial unit within the range whose value of a particular attribute is either the maximal or minimal (e.g., move to the nearest spatial unit with the maximal level of a particular resource).

A `Surviving` element abstracts aspects that govern agents' life cycle. These aspects are abstracted as `LifeResource` elements. Identified life resources are `AGE` and `ENERGY`, but support for a `CUSTOM` resource is provided. For each life resource, an agent attribute is derived. A life condition takes into account the life resource and specifies the condition that keeps the agent alive, i.e., the agent dies when such a condition stops being

met. For example, an `AGE` life resource that has an expression `'< 100'` as life condition would keep the agent alive for 100 time steps.

To allow the designer to incorporate capabilities not yet covered by the metamodel, we provided abstractions of external agents and agent capabilities, as shown in Figure 3.8. An `ExternalAgent` is an agent whose structure and behavior are entirely developed in the target simulation platform. Similarly, an `ExternalAgentCapability` is a capability developed in the target simulation platform that is assigned to the agent and activated when the simulation is running. The implementation of these elements is provided as external source code files.

Figure 3.8: Core Metamodel: External Agent and Agent Capability



As in MDD approaches for mainstream software development, the MDD4ABMS approach and its metamodel are based on a few assumptions regarding agent-based simulations. These assumptions reflect aspects inherent from the ABMS paradigm. By being inherent from the paradigm, they can be implicit on models so as to let the designer focus on other, often more relevant, simulation aspects. One main assumption adopted for the MDD4ABMS approach is that the behavior of an agent results from the activation of its capabilities at each timestep. For example, at each timestep, an agent endued with `Surviving` and `Mobility` capabilities will first activate the `Surviving` capability to determine whether it remains alive. If so, then it activates the `Mobility` capability to move throughout the environment. With respect to data collection, we assume that `OutputDataset` elements are mapped to plots in the generated simulation source code, and `Output` elements to either plot lines or bars. Other assumptions are discussed in the next sections, which present domain-specific extensions.

3.3 Domain-Specific Extensions

As previously described, in this thesis we adopted a generative, bottom-up approach for building the core metamodel. This metamodel abstracts recurrent agent-based simulation aspects; however, these aspects are very rudimentary. For example, a de-

signer who wants to endue agents with learning will have to develop such ability from scratch. Additional simulations were studied to extend the metamodel, and consequently the MDD4ABMS approach as a whole, with further aspects that increase the abstraction level of agent-based simulation models. The study is restricted to two application domains due to the expertise of the research group members and the availability of existing agent-based simulations: traffic signal control and spread of diseases. The domain analysis method followed to identify domain-specific aspects is described in Section 3.3.1. The metamodel extensions derived from the application of this domain analysis method in the two application domains are described in Sections 3.3.2 and 3.3.3.

3.3.1 Domain Analysis Method

The domain analysis method uses as input existing agent-based simulations and produces as output a list of abstractions that correspond to the domain model. The method follows from a bottom-up approach in order to reduce the bias of individual experts' views while identifying domain concepts, similar to the approach used to conceive the core metamodel.

To formulate the domain analysis method, we considered existing work in this context, focused on MAS. Hassan et al. (2009) proposed a process to guide the identification and formalization of domain concepts. A similar initiative was proposed by Garro and Russo (2010). In spite of providing valuable guidelines for identifying agents, interactions, and environmental entities, these processes do not provide support for identifying higher level concepts, such as adaptation and learning, in addition to aspects such as temporal extent, initialization, and observation. To overcome this issue, the domain analysis method adopts the ODD protocol to guide the identification and specification of most of the key characteristics of a simulation, such as its structure, agent capabilities (e.g., learning) and its underlying processes.

Our domain analysis method is composed of the following steps.

Step 1. *Concept Preliminary List.* A list of MAS-related concepts is identified using existing simulations (e.g., agents and the environment), following the steps of Hassan et al. (2009) and Garro and Russo (2010).

Step 2. *ODD-based Refinement.* The ODD protocol is used to refine identified concepts. The ODD blocks *overview* and *details* shed light on structural and simulation aspects

such as entity attributes, processes, and initialization. Details about additional agent capabilities such as learning are revealed by the *design concepts* ODD block.

Step 3. *Concept Abstractions.* Identified concepts, already refined based on the ODD protocol, are analyzed in order to find their underlying essence. Such analysis considers recurrent characteristics and behaviors. Similar concepts are abstracted as a single, essential concept, or generalized to a parent concept. In this step, a list of abstractions is built, containing the domain terminology and concepts, and generalizations.

Step 4. *Domain Modeling.* The list of abstractions is used to build the domain model.

In the following sections, we describe the metamodel extensions built by applying this domain analysis method in the traffic signal control and spread of diseases domains.

3.3.2 Traffic Signal Control and Decision-making Extensions

In the area of traffic signal control, the goal is to develop traffic control systems that i) maximize the overall capacity of the traffic network; ii) maximize capacity of critical routes and intersection that represent bottlenecks; iii) minimize negative impacts of traffic on the environment and energy consumption; iv) minimize travel times; and v) increase traffic safety (BAZZAN, 2009). In such systems, traffic signals devices (e.g., traffic lights) are used to control the flow of vehicles.

In scenarios with simple, predictable traffic demand, a fixed traffic signal control policy that produces satisfactory results can be easily developed and deployed. In contrast, in scenarios with complex traffic demands, traffic control systems should be able to *adapt* their policies to the current traffic conditions. Agent-based systems is an alternative that has been considered for creating these *Adaptive Traffic Signal Control (ATSC)* systems. Agents are autonomous and distributed by nature, and adaptive decision-making techniques such as anticipation and learning can be easily incorporated into them (BAZZAN; KLÜGL, 2013). By being endowed with learning capabilities, for example, agents can refine traffic control policies in real time and thus optimize the overall traffic flow for a more efficient use of the existing infrastructure (MANNION; DUGGAN; HOWLEY, 2016). Indeed, the successful use of agents has been reported in traffic specialized literature (CHEN; CHENG, 2010; JIN; MA, 2015). We next describe the domain analysis conducted to identify the domain concepts present in ATSC agent-based simulations.

To extend the metamodel, we considered the existing agent-based simulations

presented in Table 3.3 (column *referred as* shows how these simulations are referred henceforth). To select these simulations, we consulted agent-based simulation experts in the ATSC domain. The SOTL-Gershenson and SOTL-Cools simulations are focused on self-organizing traffic signal control agents. In the RL-Wiering, RL-Oliveira, and RL-Mannion simulations, agents adopt reinforcement learning techniques—each simulation specifies the learning task in its particular way.

Table 3.3: Simulations of Adaptive Traffic Signal Control Selected for Analysis

Simulation	*Description*	*Referred as*
Gershenson (2005) Cools, Gershenson and D'Hooghe (2013)	Simulate self-organizing traffic lights	**SOTL-Gershenson** **SOTL-Cools**
Wiering (2000) Oliveira and Bazzan (2009) Mannion, Duggan and Howley (2016)	Simulate adaptive traffic signal control agents that improve their control policies using reinforcement learning	**RL-Wiering** **RL-Oliveira** **RL-Mannion**

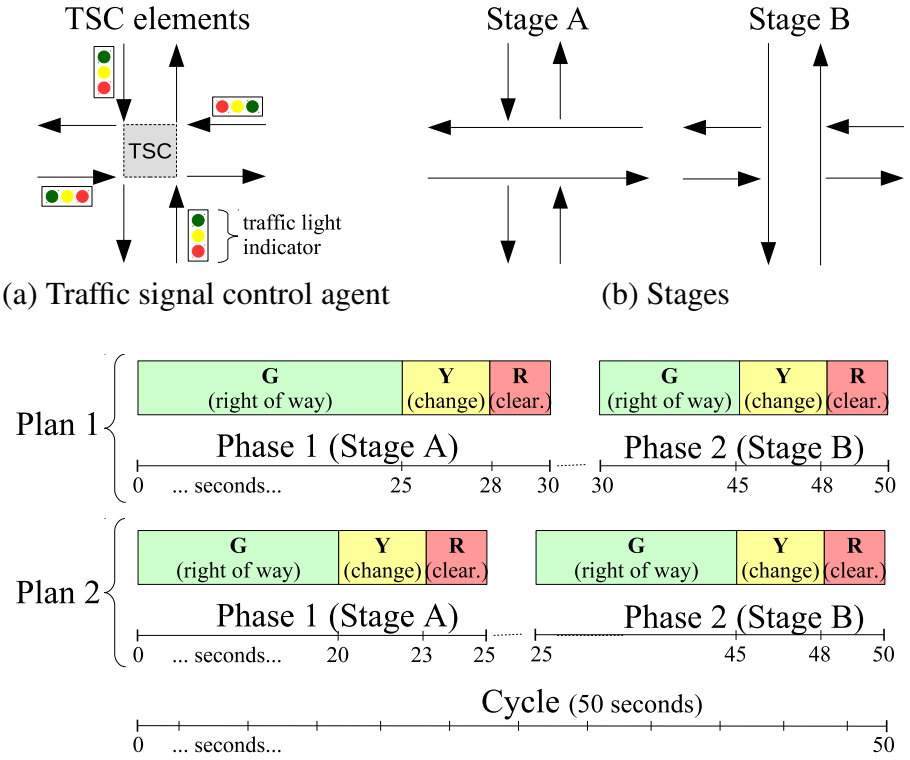
After performing step 1 of our domain analysis method, we identified different recurrent MAS-related concepts, which are: environment, agents and their perceptions, vehicles, and demand. The environment of an ATSC simulation is a traffic network, which is composed of links and nodes that represent road lanes and intersections, respectively. Such traffic network is often provided as separated files, such as open street map[3] files.

A Traffic Signal Controller (TSC) is an agent in charge of managing traffic light indicators (red, yellow, and green). TSC agents are created at each intersection and their perception is related to their incoming and outgoing lanes, such as the queue length and throughput. Additionally, it is assumed that TSC agents are able to perceive vehicle-related data, such as speed, and travel/waiting time. Figure 3.9(a) illustrates a TSC agent with its incoming/outgoing lanes and traffic light indicators.

The design of a TSC agent involves a set of concepts from the traffic control domain, which comprises our basic domain terminology and is illustrated in Figures 3.9(b) and 3.9(c). A *stage* describes a particular set of allowed traffic movements for vehicles in the lanes of the intersection. For each TSC, many stages are defined to regulate the traffic flow. A *phase* is a period in which the indicators of the corresponding stage are green, allowing the traffic flow. In addition to the green interval, a phase can specify a change interval (yellow) and a clearance interval (in which all the indicators of the intersection are red before activating the next phase). A *cycle* is a period of time during which all asso-

[3]<www.openstreetmap.org>

Figure 3.9: Adaptive Traffic Signal Control: Domain Terminology and Concepts



(a) Traffic signal control agent (b) Stages

(c) Phase, plan, and cycle

ciated stages take place. Consequently, the duration of a cycle corresponds to the sum of its phase intervals. Finally, a *plan* is a set of phases assigned to stages plus the sequence in which they are activated. An offset can be defined for a plan and corresponds to the period in which the activation of the first phase is postponed. Figure 3.9(c) illustrates two possible plans of the TSC agent, each with distinct phases. Both plans consider a cycle of 50 seconds. In order to evaluate the effectiveness of TSC agents, vehicles are created in the traffic network during the simulation according to a traffic demand.

Following step 2 of our domain analysis method, we used the ODD protocol to guide the identification of how each existing simulation deals with adaptation. In the SOTL-Gershenson and SOTL-Cools simulations, a set of adaptive criteria based on traffic conditions are introduced, i.e., a TSC agent changes to another phase according to the number of vehicles approaching and the time elapsed on the current phase.

In the RL-Wiering simulation, reinforcement learning is used by TSC agents in order to minimize the overall waiting time of vehicles. Each TSC agent has six stages and chooses the active stage based on a value function learned. Phases and plans are not considered. The value function takes into account the expected waiting times of vehicles given their destinations. In the RL-Oliveira simulation, TSC agents use reinforcement learning to learn a policy for selecting the plan that minimizes the number of stopped

vehicles over all lanes. Two phases and three plans were adopted. While one plan defines an equal duration for both phases, the other two define a longer duration for a particular phase (either north-south or east-west). The reward used was the averaged queue length at the TSC intersection. Finally, in the RL-Mannion simulation, TSC agents also use reinforcement learning in order to minimize the overall waiting time of vehicles. Each TSC agent has six stages and chooses the active stage based on a value function learned. Phases and plans are not considered. The value function takes into account the expected waiting times of vehicles given their destinations. These introduced adaptation approaches share a common characteristic: TSCs have designer-specified fixed plans, which are used as comparison baselines. Next, we describe how the metamodel was extended by following steps 3 and 4 of our domain analysis method to support adaptive traffic signal control concepts.

From the abstraction step, we observed that, essentially, a TSC is an agent that has a flow control capability for regulating the flow of a set of streams. Therefore, the metamodel was extended with an additional agent capability called `FlowControlCapability`, as shown in Figure 3.10. Such capability has a set of known streams and a set of flow regulators to manage them. These regulators can be seen as actuators of the agent. Regulators can be in certain states, such as open or closed, green or red. Additionally, regulators are homogeneous, which allows decoupling the concept of state from regulators, abstracted as actuator states. Therefore, each `ActuatorState` represents a preset that can be applied to any regulator, and its current state is the active preset. The actuator state that is automatically activated when no other state is active is the default state. Actuators can be grouped into actuator groups. All actuators of a group activate the same state simultaneously. Consequently, a group can be seen as a single actuator, and therefore we abstracted them as `Actuatable` devices. Each actuator is identified by a number that relates the actuator to the corresponding stream (i.e., a regulator 0 is in charge of regulating the stream 0, and so on). Streams are represented as an agent attribute with cardinality greater than one (i.e., a collection of streams). Additionally, we consider actuators mutually exclusive: only one actuator or group can assume a non-default state at a given moment; all others remain in the default state. Figure 3.11 illustrates how domain concepts were abstracted to these metamodel elements. In the bottom, there are concepts that were identified in steps 1 and 2 of the domain analysis. Dashed arrows point to the metamodel elements that abstract such concepts. As can be seen, a TSC agent is abstracted to an `Agent` and a `FlowControlCapability`. Each traffic signal indicator is

Figure 3.10: Traffic Signal Control Metamodel Extension: Flow Control

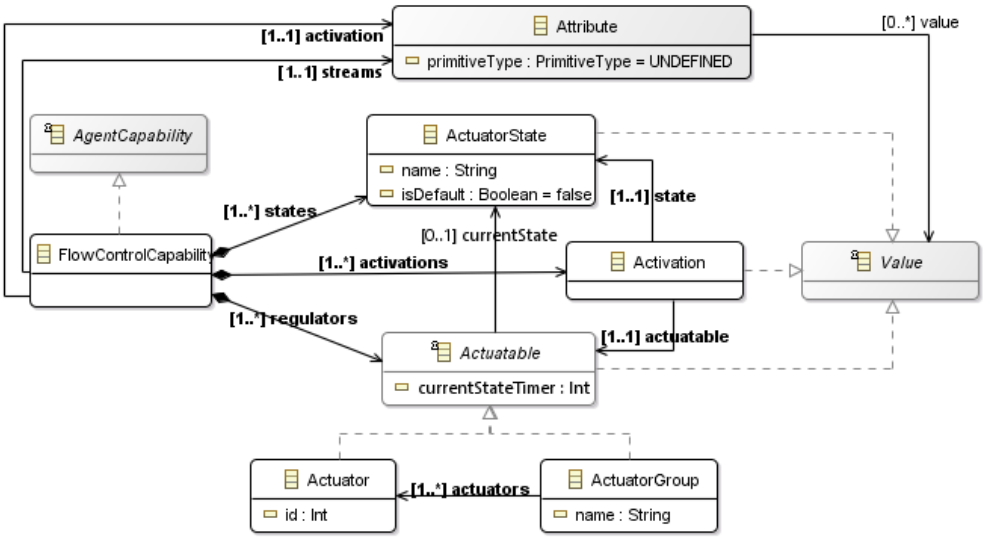
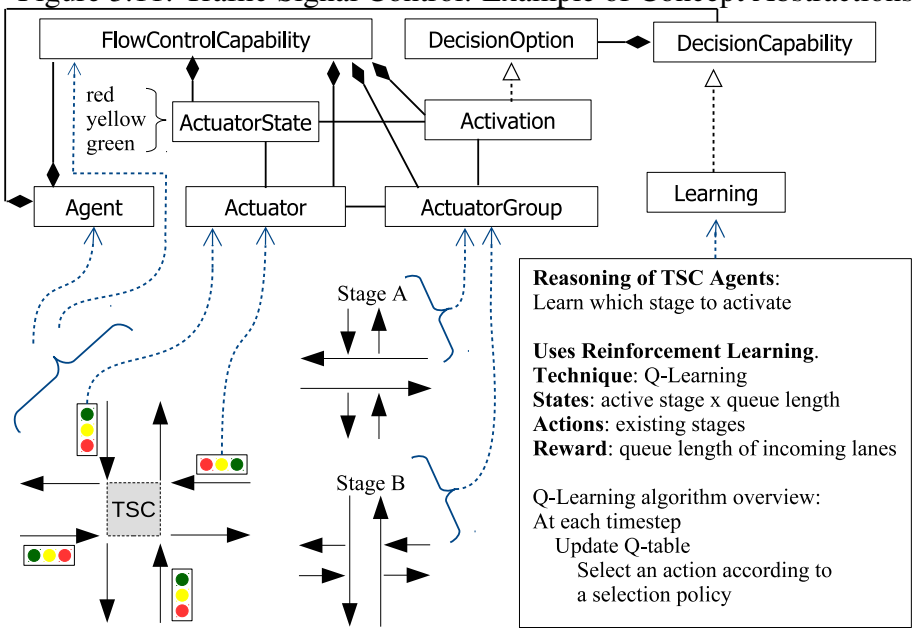


Figure 3.11: Traffic Signal Control: Example of Concept Abstractions

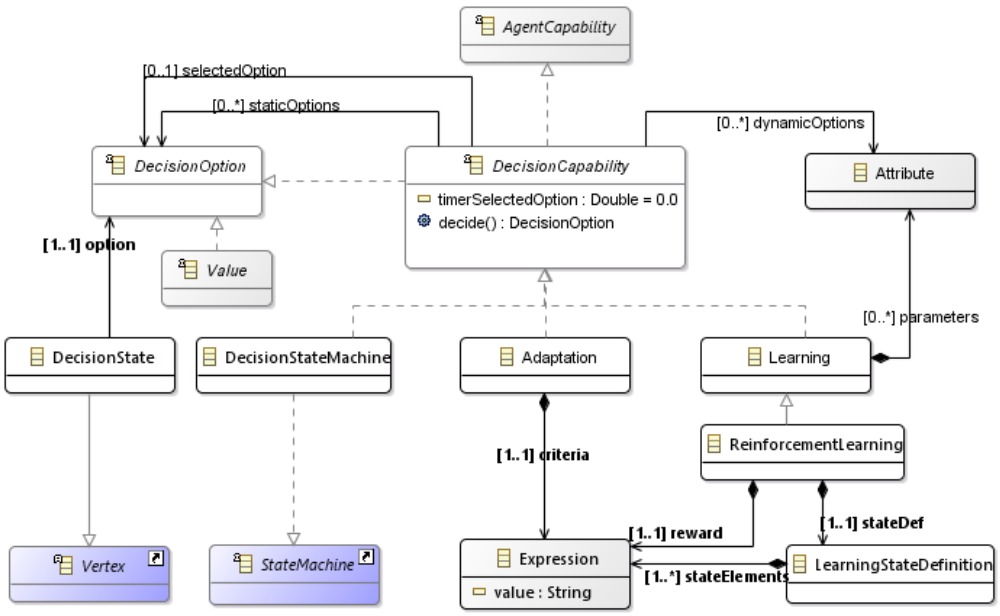


an `Actuator`, and red/yellow/green states are `ActuatorStates`. Stages are abstracted to `ActuatorGroups` given that the set of actuators that must simultaneously activate a state is obtained from stage definitions.

The behavior associated with a flow control capability is related to the management of its actuators. An agent endowed with a flow control capability selects a pair (actuator, state) for activation at every timestep. This pair is represented as an agent attribute whose value is an `Activation` element. A decision capability must be associated with this attribute. This decision capability endues the agent with the ability to select the activation that is appropriate to the current traffic conditions. Next, we describe the decision capabilities provided by our metamodel.

A decision capability represents a decision policy. Such a policy describes how to choose one amongst many available decision options. Figure 3.12 presents the elements of the metamodel that are related to decision capabilities. The set of available decision options can be either static or dynamic. Static options are those specified during the model design. Any `Value` is considered a static option. As previously shown in Figure 3.10, actuators, actuator states, and activations are specializations of `Value` and thus can be decision options. Decision capabilities can also be decision options for another decision capability. In such cases, the *id* of the decision capability is used for further reference (e.g., for specifying the states of a state machine whose options are other decision capabilities). Dynamic options exist only during the simulation execution. Therefore, we assume that these options are stored in agent attributes (e.g., a *perception* attribute).

Figure 3.12: Traffic Signal Control Metamodel Extension: Decision Capabilities



Periodically, the `decide` operation of a decision capability is activated and its output updates the value of a particular agent attribute whose update source is of type `DecisionMakingSource` (as described earlier in Table 3.1). From the domain analysis, we identified three types of decision capabilities: state machines, adaptation, and learning.

A state machine represents a fixed decision policy and is composed of states and transitions. An abstract state machine is incorporated into the metamodel, with elements such as states (vertexes), transitions, and triggers. Such abstract state machine, described in Appendix A, is essentially a subset of the UML Statemachines metamodel.[4] The metamodel customizes the following elements of the abstract state machine to define

[4]<http://www.omg.org/spec/UML/2.5/>

a state machine decision capability. The `DecisionStateMachine` extends the abstract `StateMachine` element to define a state machine that act as a decision capability, and the `DecisionState` extends the abstract `Vertex` to relate a state machine state with the decision option it represents, [5] Consequently, a `StateMachine` accepts vertexes of type `DecisionState` only, and such a constraint must be observed when the simulation model is specified.

An `Adaptation` capability represents an adaptive decision policy. There is an adaptation criterion that describes which decision option should be selected among those available—the one that meets the criterion. Such a criterion acts as a fitness or utility function and is therefore specified as an `Expression` element (see Table 3.1).

A `Learning` capability allows an agent to learn a decision policy. We consider a `ReinforcementLearning` capability, with which agents learn through experience. As agents act on the environment, they receive a reward signal based on the outcomes of previous states and actions. States are represented by a `LearningStateDefinition`, which use expressions to describe the tuple of elements that characterize a state. The concrete states are created during the simulation execution, from a cartesian product of this tuple of elements. Actions are represented by the decision options that are related to the decision capability. The reward signal is specified as an `Expression`. Finally, the set of learning parameters is represented as attributes of the learning capability. As illustration, Figure 3.11 also illustrates the abstraction of these learning concepts into a learning element. The reasoning of the TSC agents shown there is based on reinforcement learning, more specifically on the Q-Learning technique (WATKINS; DAYAN, 1992).

3.3.3 Spread of Disease Extensions

A disease is a condition of the body, or of some of its parts and organs, in which its functions are disturbed. A disease can be either infectious or noninfectious. While a noninfectious disease develops over an individuals lifetime (e.g., osteoporosis), an infectious disease can be passed between individuals (KEELING; ROHANI, 2008). Features and patterns of disease spreading are studied in the domain of epidemiology.

Mathematical models of disease spreading are available, and epidemiologists use them to analyze how an infectious disease affects a particular population of individuals

[5] In this and in all other diagrams in the following, the blue color is used to highlight elements from the abstract state machine metamodel described in Appendix A.

and which policies should be adopted to avoid or control pandemics (e.g., vaccination or quarantine). More recently, agent-based simulations have been used to analyze the spread of diseases, which is influenced by the population heterogeneity and by local, spatially bounded, interactions among individuals. When these aspects are taken into account, more effective health policies can be provided. For example, using agent-based simulation, Eisinger and Thulke (2008) showed that it is possible to control the spread of a disease over a fox population by vaccinating 10% fewer individuals in comparison to the number of vaccinations predicted by the analytical spreading model. In fact, agent-based simulations were reported as one trend in health informatics (ISERN; MORENO, 2015).

To extend the MDD4ABMS metamodel, we again used a bottom-up approach considering the simulations shown in Table 3.4. These simulations were selected from distinct repositories. On April, 2017, a query to the CoMSES model library (CoMSES Net, 2018) using disease-related terms[6] returned 12 simulations. Those on which the disease is not passed between individuals were discarded. The remaining simulations were sorted by recency. The top 3 were selected for analysis, namely: AntiVaccine, DogFox, and Product Diffusion. A query to the GIS and Agent-Based Modeling (CROOKS, 2018) repository returned the Cholera simulation. Finally, the epiDEM simulation was considered due to its potential for using with educational purposes.

Table 3.4: Simulations of Spread of Disease Selected for Analysis

Simulation	*Description*	*Referred as*
Yuan and Crooks (2017)	Simulates the impact that cyber space anti-vaccine opinion leaders can cause to the spread of a disease over non-vaccinated agents.	**AntiVaccine**
Belsare and Gompper (2015)	Simulates the spread of the canine distemper virus over a population composed of dogs and foxes.	**DogFox**
Shao and Hu (2017)	Uses spread of disease models to simulate the diffusion of product types over a social network	**Product Diffusion**
Crooks and Hailegiorgis (2014)	Simulates the spread of cholera in a refugee camp in Kenya.	**Cholera**
Yang and Wilensky (2011)	Simulates the spread of an infectious disease in a closed population.	**epiDEM**

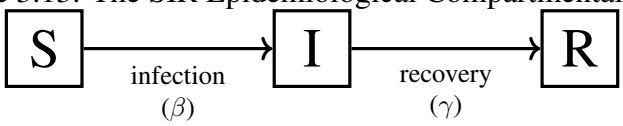
After performing step 1 of our domain analysis method, the following MAS-related concepts were identified: environment, agents that move on the environment, and the epidemiological model of disease spread. The first two concepts are already covered by our metamodel. The epidemiological model adopted is the compartmental model of

[6] "disease", "infection", "contamination", "epidemic", "immunology", and "epidemiology".

Kermack and McKendrick (1932). Next, we describe aspects of the compartmental model that are relevant to the domain analysis.

The mathematical model of Kermack and McKendrick is widely used by epidemiologists to predict and understand the spread of disease. In this model, individuals within a population are categorized into compartments. The simplest model, called SIR, adopts the following three compartments: **S**usceptible, for individuals not exposed to the disease; **I**nfected, for individuals with the disease; and **R**ecovered for individuals that have successfully cleared the disease. The dynamics of the disease is represented by transitions between these compartments, as illustrated in Figure 3.13. These transitions are governed by rates: a transmission rate β and a recovery rate γ.

Figure 3.13: The SIR Epidemiological Compartmental Model



The number of individuals in the **S** compartment changes according to Equation 3.1, which takes into account the fraction I/N[7] of the number of infected individuals I and the transmission rate β. The parameter β is a composite parameter that synthesizes a contact rate κ among individuals and the probability of transmission upon contact q, so $\beta = \kappa q$ represents the probability of transmission given contact. In agent-based simulations, inter-agent contacts occur as a result of agents' behavior and are influenced by agents' location. Therefore, only the transmission probability is adopted, so $\beta = q$. Equation 3.2 describes how the number of individuals in the **I** compartment changes. In addition to the number S of individuals that left the **S** compartment, it is affected by the recovery rate γ, the rate at which infected individuals recover from an infection, which is taken as $\frac{1}{\text{(mean duration of the diseases)}}$. Finally, the number of individuals in the **R** compartment changes according to the number I of individuals that left the **I** compartment, as shown in Equation 3.3. Individuals in the **R** compartment have lifelong immunity.

$$\frac{\partial S}{\partial t} = -\beta S \frac{I}{N} \tag{3.1}$$

$$\frac{\partial I}{\partial t} = \beta S \frac{I}{N} - \gamma I \tag{3.2}$$

$$\frac{\partial R}{\partial t} = \gamma I \tag{3.3}$$

[7] N is the population size.

Additional disease parameters can be considered to obtain variations of the basic SIR model. Deaths caused by the disease can be incorporated via a death rate μ. Temporary immunity is obtained by specifying a duration for the **R** compartment, leading to the so-called SIRS model. There are extensions to the SIR model that consider additional compartments to represent diseases with particular characteristics. A **P**assive Immunity compartment can be considered if individuals could have temporary immunity to a disease, such as when mother antibodies remain in the newborn for a period (HETHCOTE, 2000). This extended model is often called PSIR. For diseases with an incubation period, an **E**xposed compartment can be incorporated to categorize individuals that, although infected, do not manifest symptoms yet. This model is called SEIR. If temporary immunity is considered, models PSIRS and SEIRS are obtained from the latter two models.

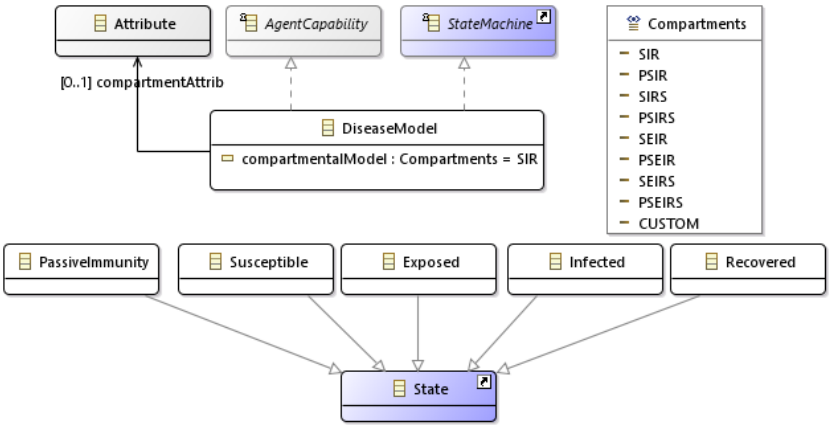
The following compartmental models are adopted in the selected simulations: SIR (epiDEM); SEIR (Cholera); PSIR (AntiVaccine); PSEIR (DogFox). The ProductDiffusion simulation adopts the SIR model, but each compartment represents a consumer state during the diffusion of new products. The **S** compartment represents consumers susceptible to product information (via the influence of product testers that diffuse product information in their online social network). The **I** compartment represents consumers that received product information, and the **R** compartment represents consumers that decided not to buy. An additional compartment, **B**, was considered in this simulation, to represent consumers that decide to buy the product.

Following step 2 of our domain analysis method, we used the ODD protocol to identify two additional aspects considered in the spread of disease simulations: interactions and initialization. The interactions specify in which agent-agent or entity-agent contacts the disease might be transmitted. For example, in the DogFox simulation, the disease can be transmitted from dogs to foxes, but not from foxes do dogs. Regarding initialization, details on how the disease start spreading are specified, as for example, the number of infected agents at the beginning of the simulation. Next, we describe how the metamodel was extended by following steps 3 and 4 of our domain analysis method in order to support all of these disease-related concepts.

The compartmental disease model is abstracted as a state machine, in which compartments act as states, and transitions between compartments act as transitions between states. A customized state machine is specified to incorporate additional, disease-related, semantics to states and transitions. Figure 3.14 shows how the metamodel was extended with a `DiseaseModel` state machine that specializes `StateMachine`. Given that the agent

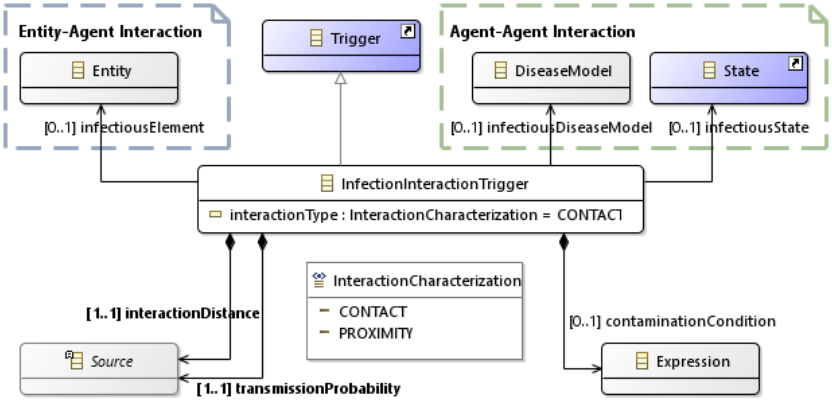
that is subject to a disease is in charge of managing its compartmental model, the disease state machine is also an `AgentCapability`. The provided compartmental models are enumerated in the `Compartments` element. One of them is assigned to the state machine to specify which compartmental model is adopted and therefore of which states the machine is composed. Specializations of `State` were specified to represent the compartments identified, and only these specialized states are accepted in the `DiseaseModel` state machine. Compartments other than the specified in the metamodel are supported by the `custom` compartmental model, which allows the instantiation of customized states and transitions. Finally, the disease state machine is associated with an agent attribute that provides access to the current compartment and is updated by the state machine.

Figure 3.14: Spread of Disease Metamodel Extension: Disease as Agent Capability



The infection caused by contact interactions is specified as a transition from the susceptible state to either the exposed or infected state. The specialized trigger `InfectionInteractionTrigger` models such contact interaction, as shown in Figure 3.15. For infections caused by disease transmission during inter-agent interactions, the trigger refers to the disease model of the other agent involved in the interaction and its infectious state. In this way, the agent may get infected whenever it is in the susceptible state and there is an interaction with another agent that is in the infectious state. For infections caused by interactions with contaminated objects, the trigger refers to the infectious entity. This kind of infection was observed in the Cholera simulation, in which agents may get infected when they interact with (ingest) contaminated water. In both cases, a contamination condition expression might be specified. If the infectious entity or agent does not meet such condition (e.g., bacterial concentration in the water below a particular threshold), the infection transition is not triggered. In the analyzed simulations, an interaction is characterized either by physical contact or by spatial proximity. In the later, there is an interaction if the distance between elements is less than a given interaction distance

Figure 3.15: Spread of Disease Metamodel Extension: Infection



(in other words, if agents are close enough). In the former, there is an interaction only if the distance between elements is zero, which means they are situated in the same spatial unit. The `interactionType` trigger attribute specifies which interaction characterization is considered by the infection transition. Finally, the transmission probability β is part of the interaction trigger, and ultimately determines whether the disease state machine will move to the transition target state when a contact interaction occurs.

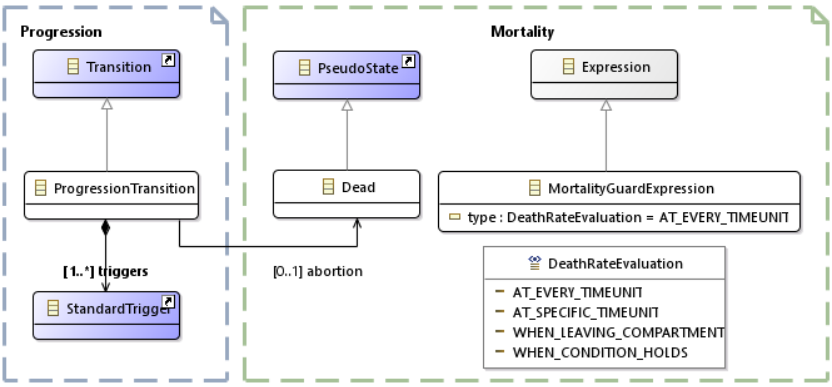
After infection, the agent' current compartment is changed according to the disease progression. The compartmental model represents this progression as transitions governed by rates. These rates are related to the duration of each disease stage, represented as compartments. In our metamodel, the duration of a particular compartment is specified by the kind[8] of the triggers associated with its outgoing transitions. Each of these transitions represents, therefore, a progression to the next compartment. To incorporate additional semantics, there is a specialized transition `ProgressionTransition`, as shown in Figure 3.16. Four distinct ways of specifying the duration of a particular compartment are provided: probabilistic, deterministic, conditional, and custom durations.

- *Probabilistic*: a rate is specified. For the cases of the **I** and **R** compartments, it is called recovery rate (γ) and immunity loss rate (w), respectively. This type of duration is specified as a trigger whose kind is *probability* and value is the rate.
- *Deterministic*: a fixed period of time τ is specified, and the agent stays in the compartment for that period. A trigger of the kind *time* is specified in this case, and its value is the fixed period of time.
- *Conditional*: a condition is specified, and the agent stays in the compartment until the condition is met. A *condition* trigger king is specified for this type of duration.

[8]The trigger kind specification is part of the abstract state machine definition (see Appendix A)

- *Custom*: a combination of the previous types. For example, the agent necessarily stays infected for a period of time and then it may recover according to a recovery rate. In this case, the `ProgressionTransition` is associated with as many triggers as the number of combined durations. The disease state machine only moves to the next compartment after all triggers have been sequentially activated.

Figure 3.16: Spread of Disease Metamodel Extension: Progression and Mortality



Deaths caused by the disease are specified as transitions to an additional pseudo-state whose kind is *final*, called `Dead`. Once the disease state machine reaches this pseudo-state, the agent dies. In all analyzed simulations, mortality is specified as death rates μ associated with the compartments in which the agent may die due to the disease. However, the circumstance under which death rates are evaluated varied among simulations. The following circumstances are provided by the metamodel.

- *At every timeunit*: the death rate is evaluated at every timestep while the agent is in the compartment.
- *At specific timeunit*: the death rate is evaluated at a particular timeunit, relative to the moment the agent has entered the compartment.
- *When condition holds*: the death rate is evaluated whenever a condition holds (i.e., when the agent ran out of energy)
- *When leaving compartment*: the death rate is evaluated only when the compartment duration has elapsed and the state machine is moving to the next compartment.

Figure 3.16 also shows the metamodel elements related to the mortality specification. Each of the first three circumstances is represented in our metamodel as a guard expression, `MortalityGuardExpression`, associated with the transition to the `Dead` pseudo-state. Depending on the circumstance, the expression value can be either a temporal value (timeunit) or a conditional expression. Additionally, a trigger of kind[8]

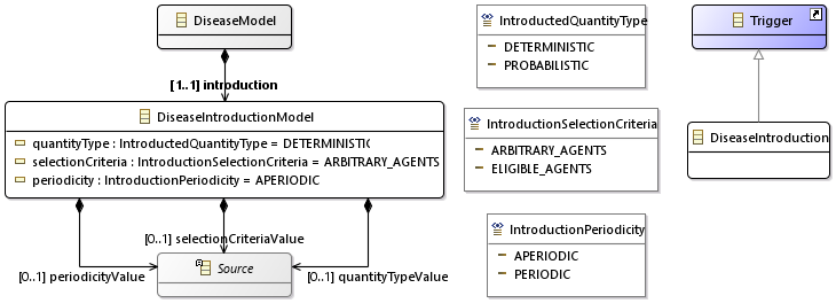
probabilistic, whose value is the death rate, is associated with this transition. For the last circumstance, *when leaving compartment*, there must exist an outgoing progression transition departing from the state in which the death rate is applied. After being triggered, which means that the disease state machine is leaving the compartment, this transition may be aborted due to the death rate. If so, the state machine moves to an `abortion` state, which is the `Dead` state. Whenever a *when leaving compartment* mortality is specified, the death rate is the guard expression of the disease progression transition.

Finally, the extended metamodel supports the specification of how the infection is introduced in the population in order to start the spread of the disease. Such an infection introduction is governed by the following aspects.

- *Quantity*: on how many entities or agents the infection is introduced. Quantity can be either *deterministic* or *probabilistic*. A deterministic quantity specifies the number of entities/agents which will be infected. A probabilistic quantity specifies the chance that any entity/agent has to be infected.
- *Selection criterion*: which entities are considered for having the infection introduced. Selection criterion can be either *arbitrary* or *eligible*. Arbitrary means that any entity/agent can be infected. Eligible means that only entities that met the eligibility criterion are considered for being infected. In both cases, only susceptible agents are considered.
- *Periodicity*: when the infection is introduced. Periodicity can be either *aperiodic* or *periodic*. In an aperiodic introduction, infection is introduced at the beginning of the simulation. In a periodic introduction, in turn, (re)introduces the infection periodically.

Figure 3.17 shows the metamodel elements related to the infection introduction. We consider that infection introduction is beyond the agent scope: it is not the agent that decides whether it is selected for being infected. If it were, the agent would have to know which others are also selected in order to respect the total number of infected agents. However, agents do not have such a global knowledge. Therefore, infection introduction is a task executed by the simulation controller. Consequently, all the aspects that govern infection introductions are specified as `DiseaseIntroductionModel` elements, which are accessed by the controller. To allow changing the compartment of the agent selected for being infected, its disease state machine has a transition from the susceptible compartment to either the exposed or infected state. A specialized trigger, `DiseaseIntroduction`, is

Figure 3.17: Spread of Disease Metamodel Extension: Disease Introduction



used with this transition. This trigger catches a signal event[9] sent by the controller to the disease state machine when the agent is selected and therefore must become infected.

3.4 Final Remarks

In this chapter, we described the main element of the MDD4ABMS approach: its agent-based simulation metamodel. The metamodel is composed of a core and domain-specific extensions. The core abstracts basic simulation aspects, such as entities, agents, the simulated environment, and the simulation execution specification. Extensions capture additional aspects that increase the abstraction level of agent-based simulation models, such as agents' abilities for decision making, flow control, and compartmental disease models. These sophisticated metamodel elements put MDD4ABMS one step forward existing MDD approaches for agent-based simulations because they consider modeling of limited aspects of simulations and leave much left to be developed in specific applications. A complete view of the agent-based simulation metamodel is shown in Appendix B.

To specify an agent-based simulation model, the designer needs to instantiate metamodel elements. In the next chapter, we present the DSL we developed to allow designers to specify simulation models.

[9] In UML a *signal* is an asynchronous communication between objects, and a *signal event* is a message requesting the reception of a specific signal. In our metamodel, the signal event notifies the state machine about the disease introduction.

4 THE ABSTRACTLANG LANGUAGE AND TOOL SUPPORT

Chapter 3 presented the MDD4ABMS core metamodel and extensions that capture aspects from the traffic signal control and spread of diseases domains. As described in the background chapter, an MDD approach combines a DSL, model transformations and, code generators to promote productivity. In this chapter, we describe the ABStractLang language for modeling agent-based simulations. A language has an abstract and a concrete syntax. The abstract syntax of ABStractLang is the MDD4ABMS metamodel. The language concrete syntax, which is specified considering the usability of ABStractLang, is presented in Section 4.1. Section 4.2 describes how model-to-code transformations are developed to enable automatic code generation from agent-based simulation models. Finally, Section 4.3 presents the ABStractme modeling tool created to allow designers to model simulations using ABStractLang and have NetLogo code generated for them.

4.1 The ABStractLang Language

To develop the ABStractLang language, we followed the systematic approach proposed by Strembeck and Zdun (2009). In this approach, four main activities comprise an iterative process for building DSLs: i) define the abstract syntax; ii) define the concrete syntax; iii) define the semantics of the language elements; and iv) integrate the language with an execution platform/infrastructure. The MDD4ABMS metamodel described in the previous chapter is used as the language abstract syntax. Next, we describe the language concrete syntax and semantics. The developed execution infrastructure for the ABStractLang language is described later in Sections 4.2 and 4.3. We adopt the following customized notation of the Backus-Naur form to specify the language concrete syntax:

- Items not enclosed with «`guillemots`» or '`quotes`' are literal (terminal) values.
- Nonterminals are enclosed with <`angle brackets`>.
- Optional items are enclosed with [`square brackets`].
- A group of items is enclosed with simple (`parentheses`).
- Items existing zero or more times are suffixed with a * (`asterisk`) symbol .
- Items existing one or more times are suffixed with a + (`plus`) symbol .
- A | (`pipe`) is used to separate alternative definitions and is interpreted as *or*.
- «`Guillemots`» denote placeholders for model data.

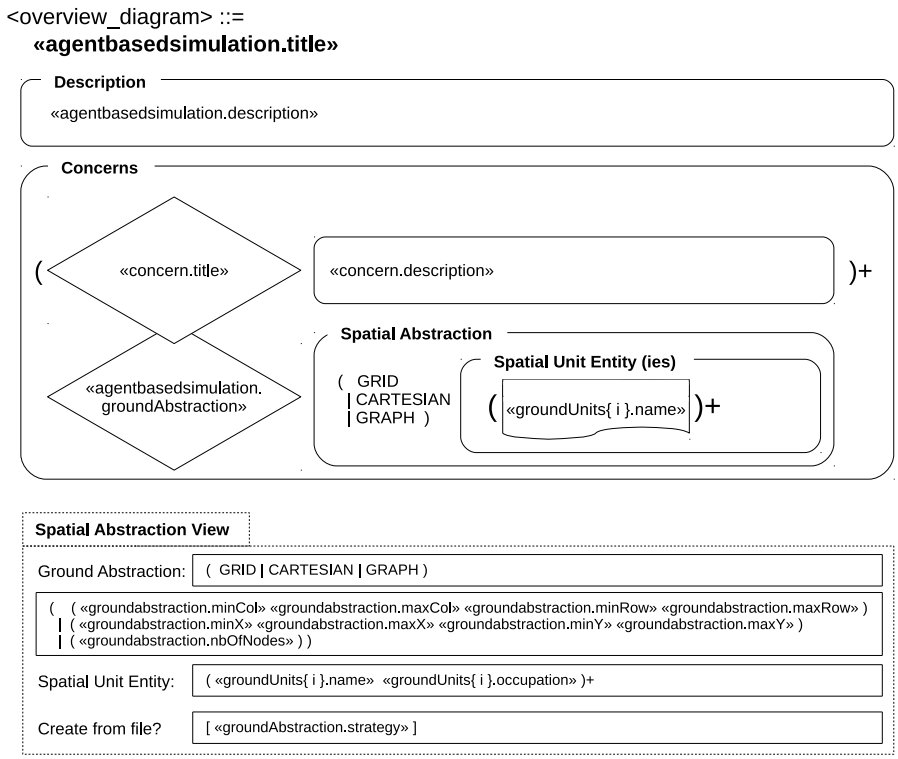
- `{Curly brackets}` denote a particular element of a set of elements.

The ABStractLang language adopts a graphical notation to present information in two dimensions (as opposed to linear, textual languages). Moreover, different shapes can be used to easily perceive different language elements (MOODY, 2009). A simulation is thus specified by means of a set of diagrams, each modeling a particular simulation concern. In the following sections, we present the two diagram types provided by ABStractLang.

4.1.1 Overview Diagram

An *overview* diagram points out the purpose of the simulation and any other essential information to provide a broad picture of the simulation being modeled. Figure 4.1 shows the concrete syntax of the *overview* diagram. It is composed of the agent-based simulation title and description, and its concerns. A diamond represents each concern, with its description beside it. If there is more than one concern, they are shown as a stack to emphasize that they represent the decomposition of the simulation into conceptual layers. Except for the bottom layer, there is no semantics associated with the ordering, and designers can choose an order using any criteria, such as the abstraction level. The con-

Figure 4.1: Concrete Syntax: Overview Diagram



cern in the bottom layer, in turn, represents the spatial abstraction of the environment in which there are situated entities. The designer chooses the appropriate abstraction among those available: either grid, Cartesian space, or graph. Then, the name of the spatial unit can be set to a meaningful term according to the vocabulary of the application area of the simulation.

ABStractLang provides views for specifying certain simulation aspects. A view is a visual element that displays information. The `Spatial Abstraction View`, shown in Figure 4.1, allows the designer to choose the spatial abstraction and configure its extent, and to set the spatial unit(s) name(s) and occupation. Finally, this view displays the creational strategy selected for the spatial abstraction, either a *GIS* or a *Geometric Data* file strategy (previously described in Table 3.2).

4.1.2 Concern Diagram

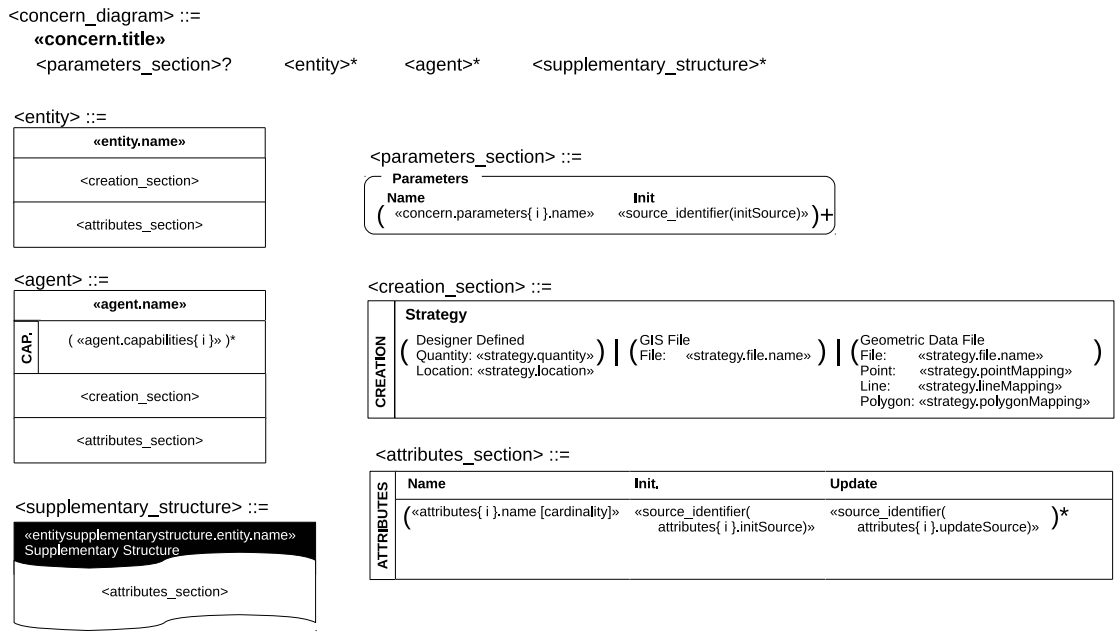
A concern diagram supports the specification of parameters, entities, agents, and supplementary structures associated with a particular aspect considered in the simulation. We group the concrete syntax description of these elements according to the MDD4ABMS metamodels. Section 4.1.2.1 describes the concrete syntax of elements from the core metamodel. The traffic signal control and decision-making metamodels have the concrete syntax of their elements described in Section 4.1.2.2. Finally, Section 4.1.2.3 describes the concrete syntax of the spread of disease metamodel elements.

4.1.2.1 Core Elements

Figure 4.2 shows the concrete syntax of a concern diagram. Considering that the modeling of a simulation can involve experts from distinct areas (e.g., traffic, humanitarian), each concern is specified in a separate diagram to allow experts to focus on concepts related to their area and avoid overloading them with unrelated elements. Additionally, by specifying a separate diagram for each concern ABStractLang deals with scalability issues of languages with graphical notation. Indeed, the partitioning of the model is recommended when engineering DSLs in order to keep models manageable (VOELTER et al., 2013).

The visual representation of parameters, entities, and attributes is inspired by the UML class diagram. Parameters are represented within a parameters section, shown

Figure 4.2: Concrete Syntax: Concern, Parameter, Entities, and Agents

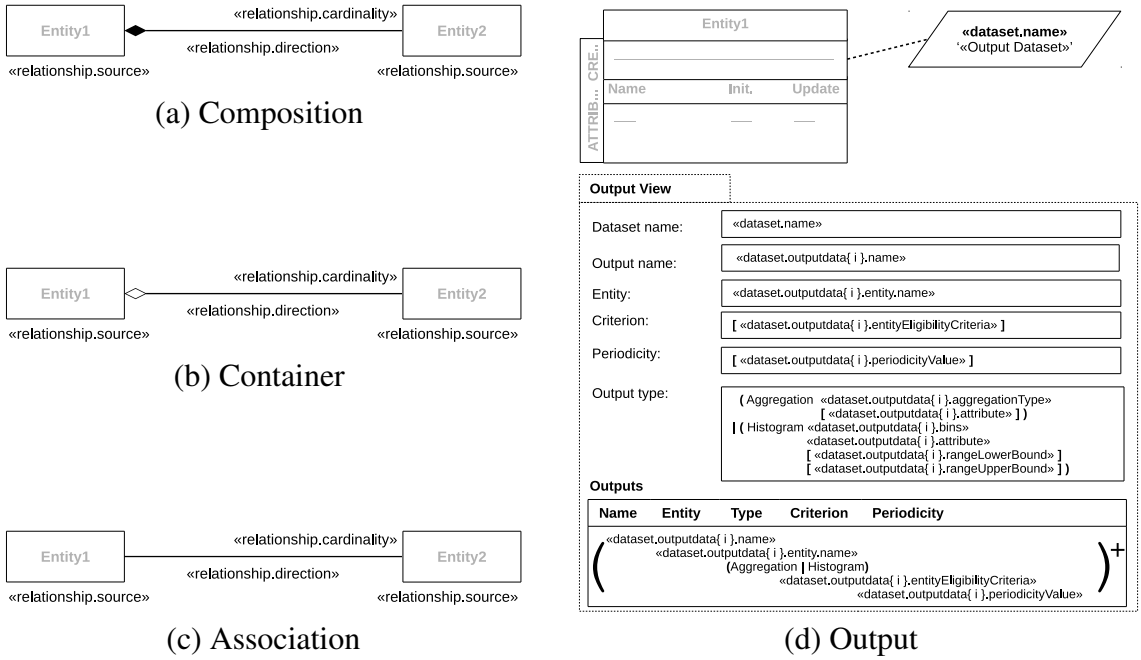


whenever a parameter is specified for the concern. For each parameter, its name and the source of its initial value are shown. Each type of source is represented by its identifier (described previously in Table 3.1), returned by the `source_identifier` function.

An entity is represented using a box with three sections. The top section shows the entity name. A creation section is in the middle, and an attributes section is in the bottom of the entity box. The creation section shows the details of the creational strategy specified for the entity, which can be either *Designer defined*, *GIS File* or *Geometric Data File*. The attributes section shows the attributes of the entity. The notation used to represent an attribute consists of its name, cardinality, the source of its initial value, and the source that updates its value during the simulation. An agent is also represented using a box with an additional section in which the agent capabilities are enumerated. A supplementary structure is represented by a rectangle with the bottom line curved to give the idea of a structure that fits in the spatial unit.

Relationships are represented with lines that connect the source and the target entity (or agent) boxes. A composition relationship, Figure 4.3(a), is represented by a line with a filled diamond at the endpoint connected to the source entity (again, inspired by UML). A container relationship, Figure 4.3(b), follows the same notation but uses an empty diamond. Finally, for an association relationship, no particular symbol is used for endpoints, as shown in Figure 4.3(c). For all these relationships, the cardinality is shown at the endpoint that is connected to the target entity, and the direction is shown below the relationship line.

Figure 4.3: Concrete Syntax: Relationships and Output



An output dataset is represented by a parallelogram connected to the entity or agent from which data will be collected, as shown in Figure 4.3(d). A view is also provided so that the designer can manage outputs, their periodicity, type (either aggregation or histogram), as well as other output details described in Section 3.2.

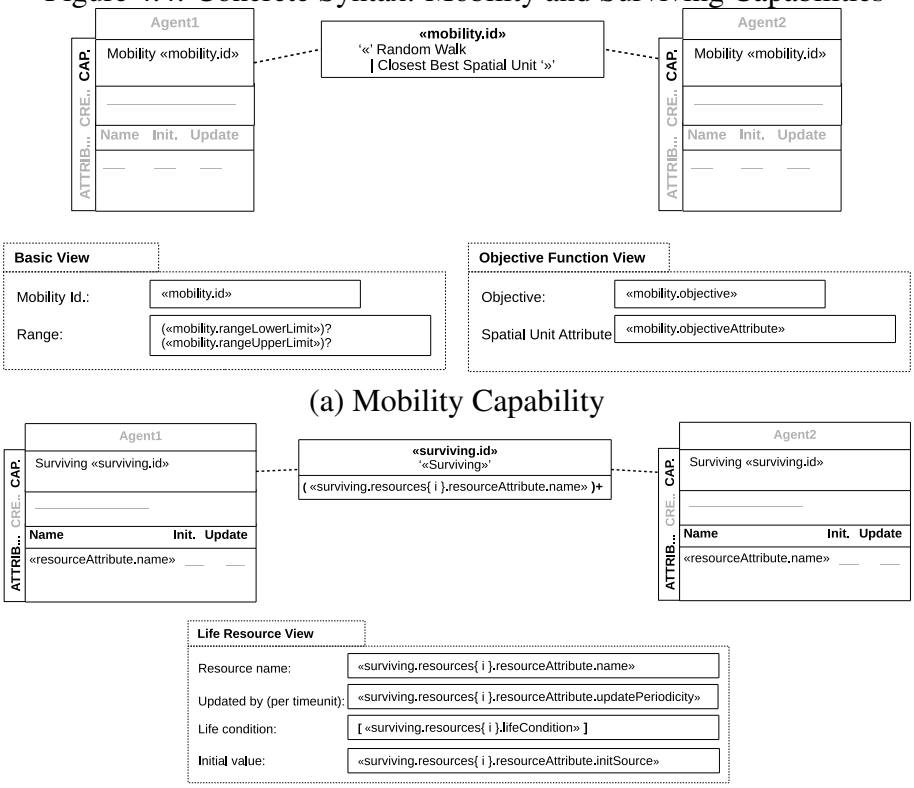
A box connected to the agent(s) that is/are endued with the capability represents a mobility capability, as well as a surviving capability, as shown in Figures 4.4(a) and 4.4(b), respectively. Additionally, the capabilities section of the agent box is changed to show the owned agent capability(ies). The agent box sections shown in gray remain unchanged.[1] For a mobility capability, the concrete syntax also defines a `Basic View` for specifying the mobility identifier and the range limits, and an `Objective Function View` for specifying the objective function, which is either to maximize or minimize a particular spatial unit attribute value. For a surviving capability, there is a `Life Resource View` to specify details of life resources, such as how the resource is updated and the condition that keeps the agent alive.

4.1.2.2 Traffic Signal Control and Decision-related Elements

The concrete syntax of a flow control capability is shown in Figure 4.5. With respect to the agent box, a flow control capability affects both the capabilities and attributes sections. Whenever such a capability is incorporated into an agent, two predefined ad-

[1] Hereafter, gray is used to denote elements whose concrete syntax remains unchanged.

Figure 4.4: Concrete Syntax: Mobility and Surviving Capabilities

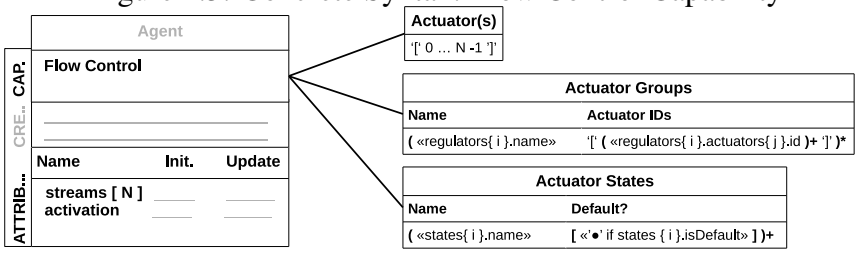


(a) Mobility Capability

(b) Surviving Capability

ditional attributes are created: `streams` and `activation` (previously described in Section 3.3.2). Additional specifications required by a flow control capability are represented as boxes connected to the agent by a line. The *actuator(s)* box shows the identifiers of actuators. Actuator groups and actuator states are shown in their corresponding boxes.

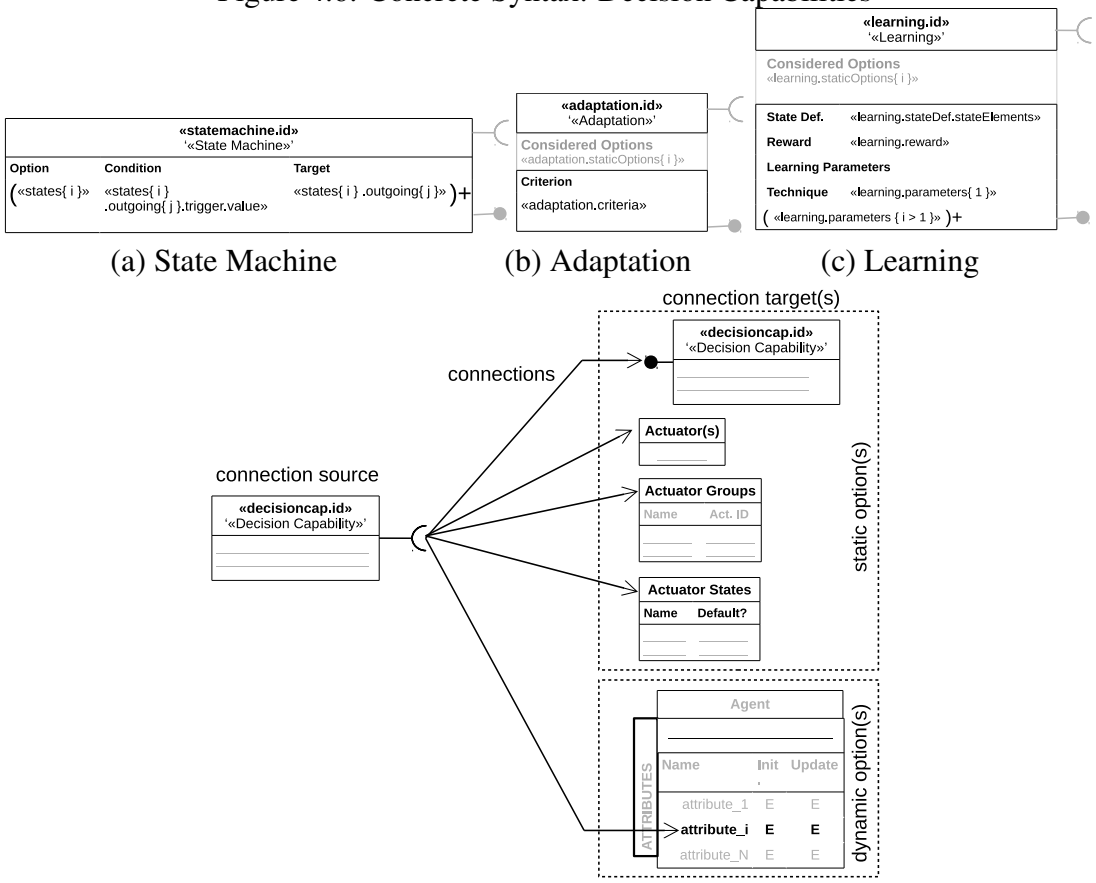
Figure 4.5: Concrete Syntax: Flow Control Capability



Decision capabilities are represented as boxes whose content describes the elements required by each capability type, as shown in Figures 4.6(a–c). The concrete syntax of each box defines, at the top section, the corresponding capability type and its identifier for further reference. The bottom section presents elements that are particular to each capability (as described in Section 3.3.2). Input (semicircle[2]) and output (filled circle) connectors, as well as other sections, are related to the specification of decision options.

[2]Semicircle and filled circle connectors are inspired by the UML component diagram.

Figure 4.6: Concrete Syntax: Decision Capabilities



Decision options are specified with *connections*, as shown in Figures 4.6(d). A capability box includes an input connector, from which a connection to any model element that represents either static or dynamic decision options can be created. As mentioned in Section 3.3.2, a decision capability can also be a decision option of another decision capability. In such a case, an output connector is shown in the decision capability box to denote that the capability is also a decision option.

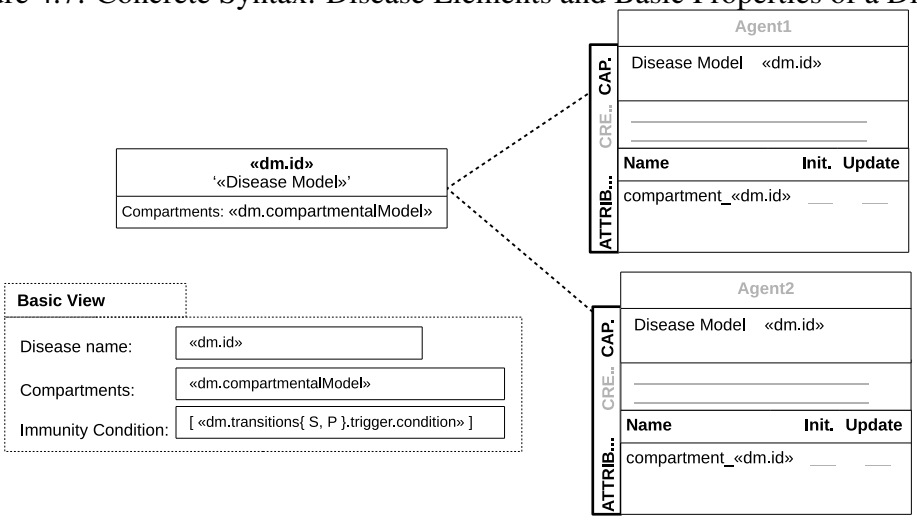
There is no restriction on the number of connections that can be created between a decision capability and model elements that represent decision options. When there is a single connection, then all the elements of the connection target are considered decision options. For example, if the target is the *actuator(s)* element (shown in Figure 4.5) then all the available actuators will be decision options of the decision capability. Similarly, if the target is another decision capability, all its possible outputs will be considered decision options. When there is more than one connection, the Cartesian product of the target elements is considered as decision options. The middle section of both adaptation and learning boxes allows specifying which options are indeed considered. It is highlighted in gray because this section is optional. If all decision options should be considered, then no

further specification is required. Otherwise, if only some options should be considered, they are enumerated in this section. Consequently, constraining the considered options is feasible only with static decision options. A state machine box does not require such a section because the considered options are those enumerated as states of the state machine.

4.1.2.3 Disease-related Elements

A disease model capability is represented as a box with the disease name at the top section, as shown in Figure 4.7. This name comes from the `id` property of the corresponding `DiseaseModel` metamodel element instance, `dm`. The bottom section presents the adopted compartmental model. A disease model specification can be shared among many agents, and a dashed line connects the disease box with all agents that are subject to it. With respect to the agent box, a disease model capability affects both the capabilities and the attributes sections (a predefined attribute is created to store the agent's current disease compartment).

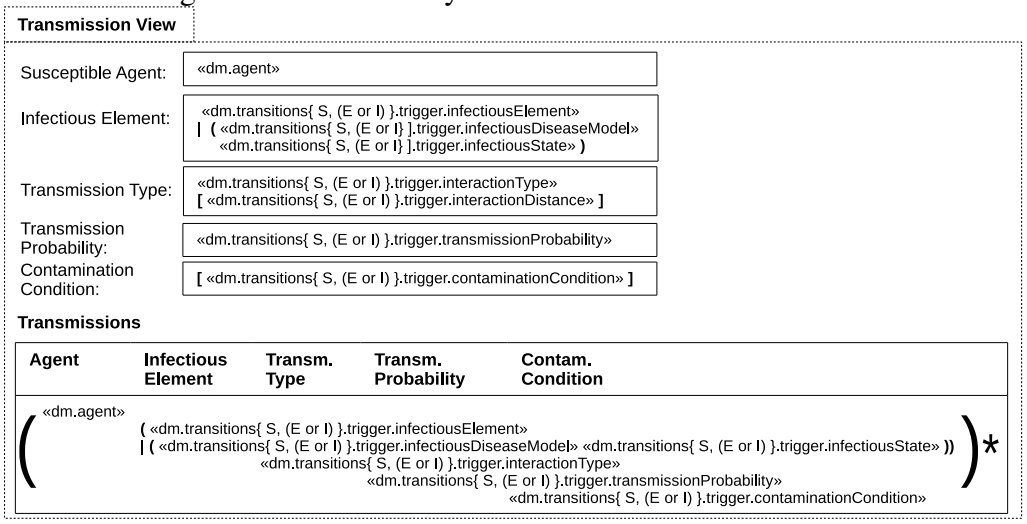
Figure 4.7: Concrete Syntax: Disease Elements and Basic Properties of a Disease



ABStractLang provides views for specifying disease-related aspects. Figure 4.7 shows the `Basic View`, which displays the following basis disease properties: disease name, and the adopted compartmental model. Optionally, an immunity condition is shown if the chosen model contains the `PassiveImmunity` compartment. Such condition corresponds to the trigger value of the transition between the `Susceptible` and `PassiveImmunity` states (a particular transition between two states is denoted by `transitions{A, B}`, where `A` and `B` are the initial letters of state machine states).

Transmissions due to interactions are presented in the `Transmissions View`, shown in Figure 4.8. In its top section, this view displays the specification of a particular

Figure 4.8: Concrete Syntax: Disease Transmission View



transmission. As previously described in Section 3.3.3, the transmission specification requires specifying an interaction between a susceptible agent and an infectious element, in addition to the transmission type (either contact or proximity), transmission probability, and contamination condition (optional) that are considered in the interaction. The bottom section of the `Transmissions View` displays, in a tabular fashion, all the transmissions already specified for the disease model, one per line.[3] ABStractLang provides additional views for specifying disease progression, mortality, and infection introduction. These follow the same notation of the `Transmissions View`, and therefore are presented in Appendix C. Examples of the concrete syntax described in this section are shown later in Section 4.3.

4.2 Model-to-Code Transformations

While the ABStractLang provides the support needed to model agent-based simulations, support for code generation is fundamental to reduce the effort to develop them. The generated code should include all the variables and operations that implement an agent's behaviors and capabilities, in addition to the specified entities and the setup and execution of a simulation.

To generate code from a simulation model and thus exploit the benefits of an MDD approach for agent-based simulations, we specified model-to-code transformations to generate code for the NetLogo simulation platform. Model-to-code transformations are

[3] The value of the i-th column is shown as the $i-$th row in the figure due to space constraints. All these data constitute a single row, actually.

performed through the use of production rules, which transform instantiated concepts of our metamodel to NetLogo code statements and blocks, and preserves the semantics of the model elements..

Generated code for entities and agents must include statements for defining their attributes, to create them according to the specified creational strategies, and to initialize and update attribute values according to the specified sources. The agent source code must include additional statements to implement the agent behavior, which is obtained by activating each of its capabilities at each timestep as detailed previously in Section 3.2. For each capability, an initialize operation (code block) is generated to set it up. For decision capabilities, in addition to operations that are particular to each of them, a `decide` operation is generated to select and return the appropriate decision option. For the flow control capability, an operation to select and apply an activation (actuator-state pair) is generated. For a disease model capability, all the source code that implements the disease state machine (states, transitions, and triggers) is generated.

The generated simulation source code also includes statements for defining the model parameters, which are coded as attributes of the simulation. If the value of these parameters is provided by the user, operations to create input components are generated. A `setup` operation is generated to combine the creation and initialization of all the entities and agents, and a `go` operation is generated to combine the update of entities, agents, and the environment at every timestep, as well to activate the agents' behavior.

Production rules for the MDD4ABMS approach are specified and documented using the Xpand template language.[4] Each Xpand template describes the source code that is generated for its corresponding metamodel element. To illustrate how the developed production rules work, Table 4.1 describes, using natural language, a subset of the rules for transformation of agents, their attributes, and flow control and learning capabilities (more specifically for the Q-Learning (WATKINS; DAYAN, 1992) technique). The production rule column indicates the rule name which, when applied, is transformed into the content presented in the transformation column. Model elements are enclosed with «guillemots». The meaning of NetLogo statements shown in this column is as follows: `breed` and `breed-own` statements are used to declare an agent type and their attributes, respectively; `procedure` and `reporter` are used to declare operations—the latter declares an operation with return value; and `set` is the assignment statement.

Usually, agent capabilities demand specific data structures. For example, the Q-

[4] <http://www.eclipse.org/modeling/m2t/?project=xpand>

Table 4.1: Subset of the Production Rules

Production Rule	**Transformation**: Metamodel Element(s) → to NetLogo Statement(s)
agent type **agent attributes**	For each «`Agent`» → `breed` For each «`agent.attributes`» → `breed-own` For each «`agent.capabilities`» → `breed-own` for derived attributes (e.g., Q-table)
flow control capability	For each «`agent.capabilities`» of type *FlowControlCapability* → **flowcontrol initialize** **flowcontrol select activation**
flowcontrol initialize	For each «`regulators`» → `set` statements for setting up actuator/groups For each «`states`» → `set` statements for setting up actuator states For each «`activations`» → `set` statements for setting up activations
flowcontrol select activation	`set` statements for selecting an appropriate activation and for applying it on the corresponding «`regulators`»
reinforcement learning capability	For each «`agent.capabilities`» of type *ReinforcementLearning* → **qlearning initialize** **qlearning compute reward** **qlearnig learn** **qlearning decide**
qlearning initialize	For each «`stateDef.stateElements`» → `set` statements for setting up states For each «`staticOptions`» and «`dynamicOptions`»→ `set` statements for setting up actions
qlearning compute reward	For the «`reward`» expression → `reporter`, which evaluates the reward expression and returns its value
qlearnig learn	`procedure` that: Invokes reward `reporter` to compute its value Defines `set` statements for updating the `Q-table` using the Q-Learning update rule: $Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$ where: Q is the Q-table s, s' are the current and resulting states a, a' are the current and resulting actions r is the reward α and γ are learning «`parameters`».
qlearning decide	`reporter` that: Invokes the **qlearning learn** `procedure` Selects and reports an action according to a selection policy (for an ϵ-greedy policy, it would be $\mathrm{argmax}_a Q(s,a)$ with probability $1-\epsilon$, and a random action with probability ϵ)

Learning technique demands a data structure (called *Q-table*) for storing the discounted expected rewards of the agent. Such data structures are derived from each particular agent capability and are generated automatically by the production rule **agent attributes**. The production rules **flow control capability** and **reinforcement learning capability** make use of extra rules that produce code related to their operations. As can be seen from the rules that produce a reinforcement learning capability, all the source code that implements the capability behavior is generated automatically.

To guarantee that generated simulations are adequately coded (e.g., the QLearning technique) and correctly implement the expected behaviors, a verification procedure was performed, as recommended by Crooks and Hailegiorgis (2014) and Iba, Matsuzawa and Aoyama (2004). Such a verification consisted of detailed inspections of the source code and debugging sessions, to ensure that all the units of code are performing their corresponding operations and are correctly integrated to implement each agent capability. Additionally, unit tests were conducted to assert the correct code generation from different parts of our metamodel. Existing NetLogo simulations were specified using the ABStractLang language, and the generated simulation code was executed to validate the produced outputs.

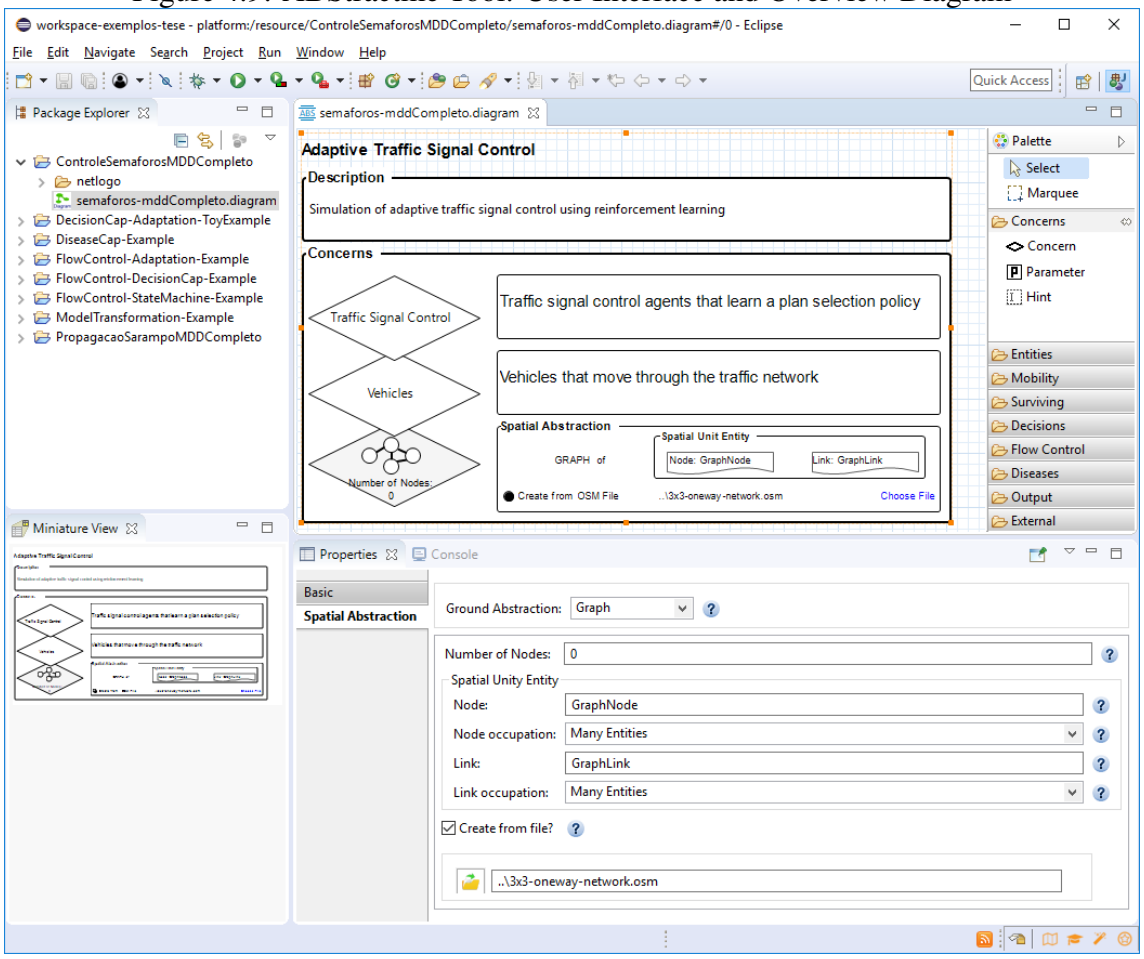
4.3 ABStractme Modeling Tool

The ABStractme modeling tool allows specifying agent-based simulations using the ABSstractLang language. The tool is an Eclipse plugin, and provides automated code generation via the execution of the Xpand production rules. ABStractme is built upon Graphiti[5], an Eclipse framework for developing graphical diagram editors. Constraints related to both the agent-based simulation metamodel and the ABStractLang concrete syntax are validated by the ABStractme tool at runtime, to avoid the specification of inconsistent simulation models (e.g., invalid connections to decision options, or or invalid disease model states).

Figure 4.9 shows the graphical interface of the ABStractme tool, which is composed of four main parts. On the left-hand side, the *Project Explorer* shows the projects created by the designer and allows managing projects and diagrams, as usual in the Eclipse platform. The modeling elements of the ABStractLang language are provided on the right-hand side, as a *Palette* of elements from which the designer can drag and drop them into the diagram. The *Properties* section on the bottom shows the view (visual element that displays information, as described in Section 4.1) related to the model element that is currently selected in the diagram. Finally, the *Diagram Editor*, on the center, is the main part of ABStractme, in which the designer specifies the simulation elements. In the diagram editor shown in Figure 4.9, there is an overview diagram for an adaptive traffic signal control simulation. Two concerns are specified, in addition to the spatial abstraction concern (a graph created from an open street map file).

[5]<https://eclipse.org/graphiti/>

Figure 4.9: ABStractme Tool: User Interface and Overview Diagram



When the designer double-clicks a concern element, ABStractme opens the corresponding concern diagram. Figure 4.10 shows the diagram of the *Traffic Signal Control* concern, which specifies a traffic signal controller agent. The agent is endued with a flow control capability to manage the traffic flow at its intersection. Two actuator states (`green` and `red`—default) are specified. Because there are only two actuators (one for each flow direction, either north-south or west-east), no actuator group is specified. The `activation` attribute is updated periodically according to a decision making source (detailed in Section 3.2) that points to the `plan learning` capability. Such learning capability uses *Q-learning* to learn a policy for selecting the plan that minimizes the number of stopped vehicles over all its incoming lanes. The reward used is the number of vehicles waiting at the intersection. The decision options of the learning capability are the three signal plans. Signal plans are specified as state machines because they are fixed decision policies for selecting an activation according to the elapsed time. Each plan specifies the duration of two phases: north-south and west-east. Plan `alpha` gives equal green phase duration time for both phases. Plans `beta` and `gamma` give priority to the horizontal and

vertical directions, respectively. The decision options considered by these state machines are combinations of actuators and actuator states, as can be seen from the connections. Transitions between states are based on time. For example, in the `gamma plan` state machine, state `(0, green)` is activated for 45 seconds and `(1, green)` for 15 seconds.

From the execution of the production rules that generate code, NetLogo source is automatically generated by the ABStractme tool. From the simulation model shown in Figure 4.10, the generated code includes all the required statements to set up and run traffic signal controller agents. To illustrate the output of the transformation rules, Appendix D presents fragments of the NetLogo source code that are generated from this simulation model, in particular for setting up and running a simulation, and for the state machine `gamma plan` and the reinforcement learning `plan learning` capabilities.

Figure 4.11 shows a concern diagram of a spread of disease simulation. Two agents, `Native` and `Immigrant`, are subject to the `Measles` disease. Different transmission rates β are specified for each agent-agent interaction, as shown in the *properties* view. A `Random Walk` mobility is shared by the agents. For each agent, the simulation specifies an `Output Dataset` to collect the number of susceptible, infected, and recovered agents. Finally, two `External Agent Capability` elements are specified to color each agent according to its current disease state.

4.4 Final Remarks

In this chapter, we described the concrete syntax of ABStractLang, our domain-specific language for creating agent-based simulation models. The language adopts a graphical notation and provides building blocks to instantiate elements of the MDD4ABMS metamodel. We also described how model-to-code transformations were developed in order to generate NetLogo source code from models. Finally, we described ABStractme, a modeling tool created to allow designers to model agent-based simulations using the ABStractLang and have its NetLogo source code generated automatically. The tool was developed as an Eclipse plugin and is available online.[6] In the next chapter, we detail how the MDD4ABMS approach was evaluated in order to demonstrate whether it indeed provide benefits for developing agent-based simulations.

[6] <http://www.inf.ufrgs.br/prosoft/projects/mdd4abms/>

Figure 4.10: ABStractme Tool: Concern, Flow Control, State Machine, and Learning

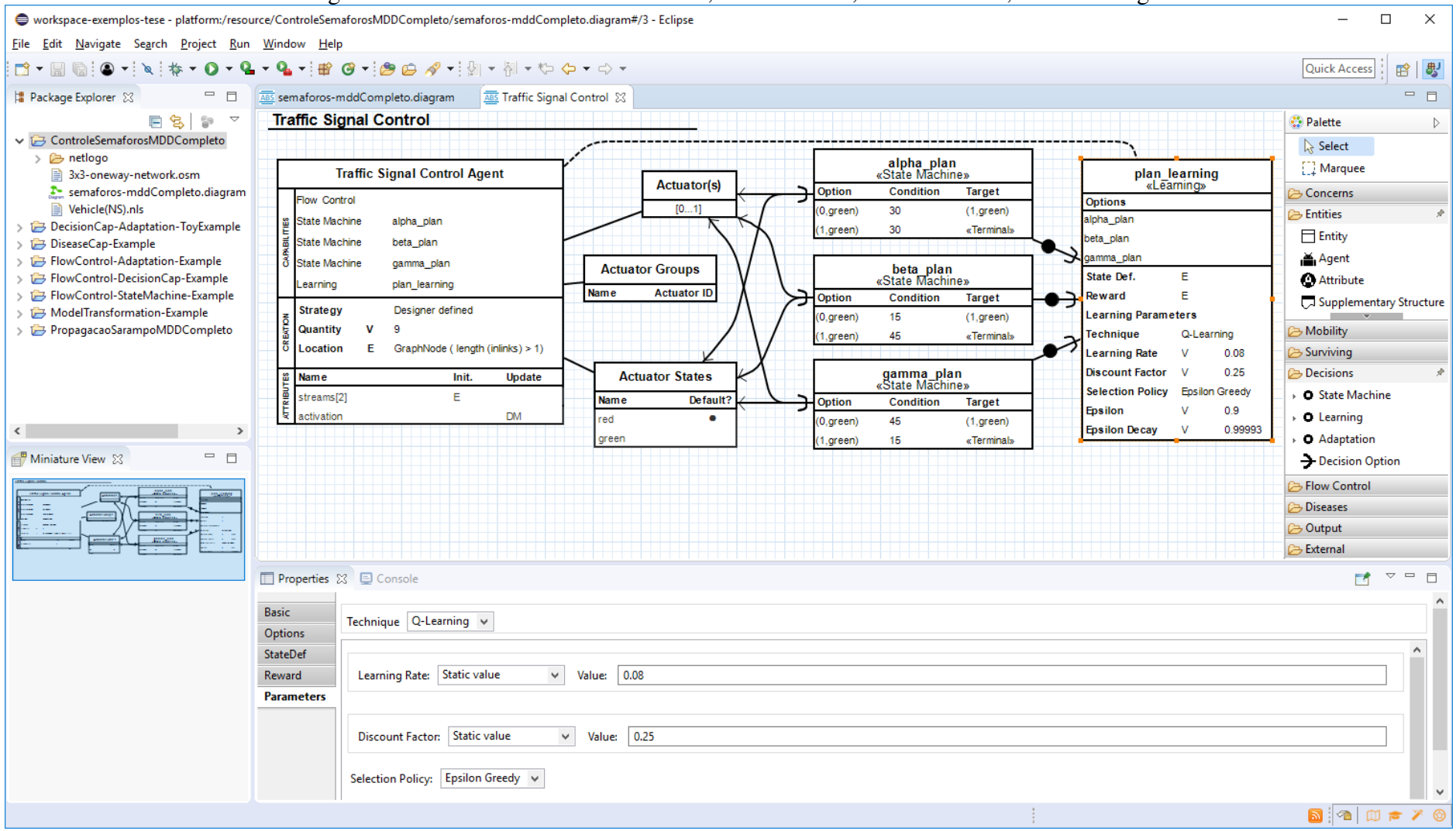
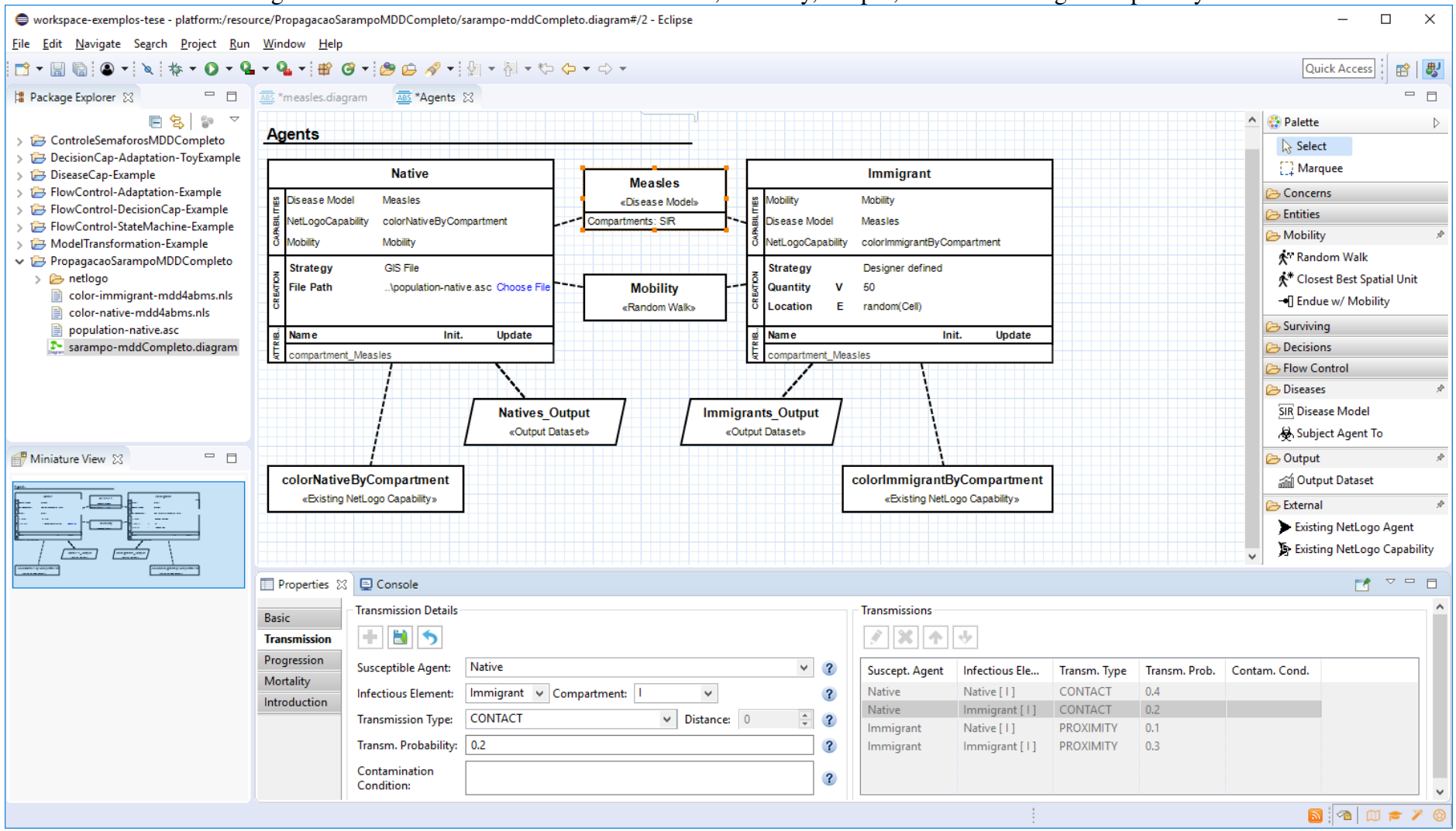


Figure 4.11: ABStractme Tool: Disease Model, Mobility, Output, and External Agent Capability



5 EVALUATION

In Chapters 3 and 4, we presented MDD4ABMS, an approach for model-driven agent-based simulation development that is composed of (i) a core metamodel with extensions, (ii) a modeling language, and (iii) model-to-code transformations to generate code for the NetLogo simulation platform. In this chapter, we evaluate MDD4ABMS through empirical studies. In Section 5.1, we describe a user study conducted to evaluate the ability of participants to comprehend agent-based simulation models specified with the ABStractLang language. Next, Section 5.2 presents an empirical study in which a software engineering metric is used to assess the productivity of using MDD4ABMS to develop agent-based simulations. Finally, Section 5.3 describes a user study that assesses the benefits provided by MDD4ABMS, with respect to design quality and development effort, as it is used by developers with little expertise in ABMS to develop simulations in the adaptive traffic signal control and spread of disease domains.

5.1 User Study: Core Metamodel and the ABStractLang Language

In order to evaluate whether the MDD4ABMS core metamodel and the ABStractLang language ease the understanding of agent-based simulations, a user study was conducted. The study assesses the ability of participants to understand, or comprehend, agent-based simulation models instantiated from the abstractions provided by the metamodel and modeled with the ABStractLang language. The evaluation is based on the premise that comprehension is one of the key goals of software models and affects the design, implementation, and evolution of software systems (MOHAGHEGHI; DEHLEN; NEPLE, 2009).

The study was designed following the *Goal-Question-Metric* (GQM) template (BASILI; SELBY; HUTCHENS, 1986), which is recommended in literature for conducting experiments in software engineering (FENTON; BIEMAN, 2014; WOHLIN et al., 2012). Following the GQM template, the goal of the study is: *to assess the benefits provided by ABStractLang for modeling the simulated environment and entities of agent-based simulations, evaluate the ABStractLang from the perspective of the researchers, as it is used by graduate and undergraduate students to comprehend implemented agent-based simulations in a multi-project study*. The study was conducted in an early stage of this thesis to collect empirical evidence to support work towards a fully featured MDD

approach for ABMS. In that stage, abstractions for agents, their capabilities, as well as simulation outputs were not yet incorporated into the metamodel. Consequently, the study considers modeling of only the simulated environment and its entities.

The following two research questions are derived from the study goal:

RQ1. Does ABStractLang *increase the correct comprehension* of the *structure* and *execution specification* of implemented agent-based simulations?

RQ2. Does ABStractLang *decrease the time* to correctly comprehend the *structure* and *execution specification* of implemented agent-based simulations?

To answer these research questions, we selected the following three metrics:

M1. *Score* of comprehension of agent-based simulation models. (RQ1)

M2. *Time* spent to comprehend agent-based simulation models. (RQ2)

M3. *Degree* of perceived ease of comprehension. (RQ1)

Metric M1 indicates the amount of comprehension of the simulation model by participants, while M2 measures the time spent to comprehend such models. Metric M3 captures the degree at which participants believe that ABStractLang facilitates the comprehension of models. Next, we describe the procedure adopted to conduct the study and collect these metrics.

5.1.1 Procedure

In this study, we compare models produced with ABStractLang and code written (referred to as models) in the NetLogo platform. Although there are simulation platforms based on graphical languages —e.g., SeSAM (KLÜGL; HERRLER; FEHLER, 2006)— they were not selected for comparison because they do not provide support for using external files (such as GIS) and, therefore, its models are not semantically equivalent to models specified with our metamodel and the ABStractLang language. Moreover, advantages of NetLogo over other simulation platforms have already been reported in literature (RAILSBACK; LYTINEN; JACKSON, 2006).

Participants in our study had to answer an assessment questionnaire about agent-based simulations modeled with ABStractLang as well as implemented in NetLogo. The questionnaire had twelve questions associated with elements of the environment specification (spatial abstraction, model parameters, and entities) and execution specification

Table 5.1: Core Metamodel and ABStractLang Study: Summary of the Questionnaires

#	**Purpose: identification of...**	**Model Element**
Q01	the spatial abstraction	Spatial Abstraction
Q02	model parameters	Parameters
Q03	parameter data types	
Q04	*the source of initial parameter values*	
Q05	entities in the model	Entities
Q06	entity attributes	
Q07	entities that are owners of given attributes	
Q08	attribute data types	
Q09	*how attributes are initialized*	
Q10	*how entities are created, and how many*	
Q11	relationships between parameters and entities	
Q12	*the initial location of the created entities*	

(the way parameters are initialized and details of entity creation). Table 5.1 presents the purpose of each question and the modeling element to which it is related. Questions highlighted in *italics* (Q04, Q09, Q10, and Q12) are those associated with simulation execution specification.

Participants were asked to evaluate both alternatives (ABStractLang and NetLogo) considering two agent-based simulations: Haiti Earthquake (CROOKS; WISE, 2013), and TELL ME Influenza (BADHAM; GILBERT, 2014). The comprehension score of a participant (metric M1) is calculated based on the correctness of each answered question. The score of each question is a value between 0 and 1, according to the fraction of correctly answered sub-questions. For example, Q3 asks for listing three simulation parameters, so if only two are provided correctly, the score is 0.66. Participants were allowed to leave blank answers, in case the answer was not found. These were considered mistakes. A survey tool (Lime Survey) was used to collect the time it takes a participant to answer the questionnaire (metric M2). Finally, to measure the perceived ease of comprehension (metric M3), each participant had to fill a post-test form in which they were asked to provide a score for the perceived ease of comprehension.

5.1.2 Participants

The study involved 26 volunteer participants, graduate and undergraduate students in Computer Science, who performed the study activities in a laboratory. Each participant was exposed to the two techniques (ABStractLang and NetLogo), but with distinct sim-

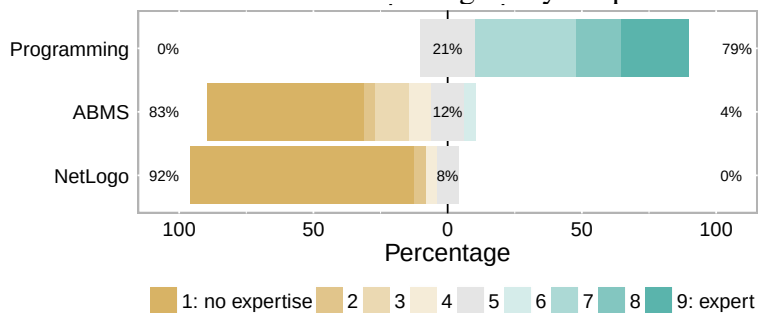
ulation models (Haiti and TELL ME) to avoid model bias. In order to avoid carryover effects, each participant was randomly assigned to one of four treatment groups[1] and then measured in two consecutive sessions, in which the treatments order was changed. The treatment conditions of each group are shown in Table 5.2. The choice for the groups in which six or seven participants were assigned was completely random. A two-hour slot was estimated for the participants to complete both sessions, but there was no time limit.

Table 5.2: Core Metamodel and ABStractLang Study: Treatment Groups

		Session 1		Session 2	
Group	**N**	**Model**	**Technique**	**Model**	**Technique**
A	7	Haiti	ABStractLang	TELL ME	NetLogo
B	6	TELL ME	ABStractLang	Haiti	NetLogo
C	7	Haiti	NetLogo	TELL ME	ABStractLang
D	6	TELL ME	NetLogo	Haiti	ABStractLang

Before the experiment, the participants were given a 10-minute explanation in which we explained the concepts related to ABMS and the experimental procedure. In addition to this talk session, each participant was asked to fill a background form. Regarding demographic characteristics of participants, 92.3% were male. 65.4% reported age between 20–25. Education was reported as undergraduate (42.3%), master (38.5%), and PhD (19.2%). We also asked participants to quantify their expertise in Programming, ABMS, and NetLogo. Results are summarized in Figure 5.1. The expertise is a value ranging from 1 to 9, where 1 means no expertise and 9 means high expertise.

Figure 5.1: Core Metamodel and ABStractLang Study: Expertise of Participants



As it can be seen from the expertise plot shown in Figure 5.1, all the participants reported programming expertise above the intermediate level, which means they would not have problems to understand the logic behind a NetLogo source code—considering that its code describes elements and types in a structured textual way similar to other

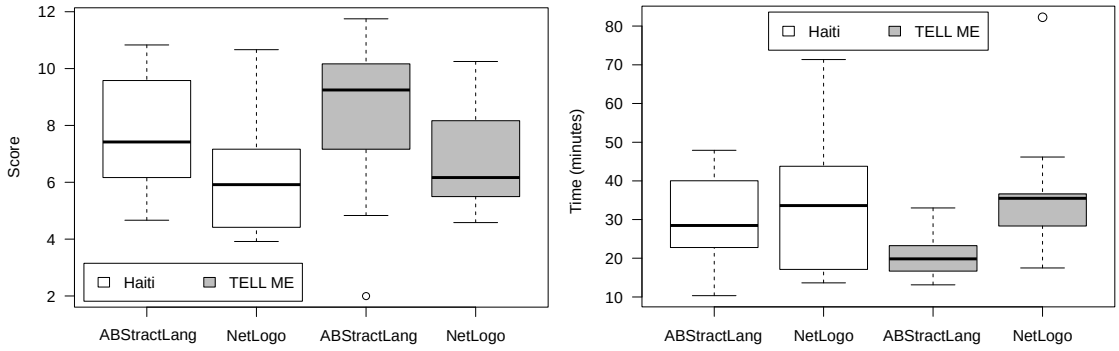
[1] A set of participants that have received the same treatment, i.e., that have developed a particular simulation with a particular technique.

programming languages, and some statements are indeed equal, such as `if` and `for`. Expertise in ABMS was reported as below the intermediate level by 83% of the participants. Finally, expertise in NetLogo was reported above the intermediate level from 92% of the participants and, from these, 85% reported no expertise. Therefore, no participant needed to be excluded due to prior advanced knowledge in NetLogo.

5.1.3 Results and Discussion

Results are summarized in Figure 5.2 and detailed in Table 5.3. It can be seen that ABStractLang achieved better results for our two measurements, in both scenarios. In the Haiti simulation, participants using ABStractLang achieved, on average, a score that is 1.57 point higher than using NetLogo, taking less time (3.58min, on average). In the TELL ME simulation, the score difference is smaller, 1.47 on average, but still positive. However, participants using NetLogo took much more time (15.8min, on average).

Figure 5.2: Core Metamodel and ABStractLang Study: Summary of Score and Time



(a) Overal Score (b) Overal Time

Table 5.3: Score and Time to Comprehend Simulations

Metric	Simulation	Technique	Mean	SD	Min	Max
M1: Score (0..12)	Haiti	ABStractLang	**7.80**	2.13	4.66	10.83
		NetLogo	6.23	2.06	3.91	10.66
	TELL ME	ABStractLang	**8.35**	2.74	2.00	11.75
		NetLogo	6.88	1.77	4.58	10.25
M2: Time (minutes)	Haiti	ABStractLang	**29.72**	12.18	10.32	47.91
		NetLogo	33.30	17.96	13.64	71.37
	TELL ME	ABStractLang	**21.08**	6.49	13.12	33.02
		NetLogo	36.88	15.55	17.50	82.29

With respect to *time*, a Kruskal-Wallis test[2] revealed a significant difference among the groups ($\chi^2 = 10.233$, $p = 0.0167$). A post hoc Dunn's test with Holm correction showed significant differences between ABStractLang and NetLogo in the TELL ME simulation ($p = 0.011$). By using ABStractLang, the average comprehension time is 42.8% lower than by using NetLogo. However, results show no significant difference between ABStractLang and NetLogo in the Haiti simulation. We believe such differences are due to the complexity of each simulation. We consider TELL ME more complex because it specifies more entities, attributes, and model parameters than Haiti. Given that participants had previous expertise in programming, understanding a simple simulation by using NetLogo may be as easy as by using ABStractLang. On the other hand, for complex models with complex entities, many parameters, and many setup routines, the comprehension time increases by using NetLogo.

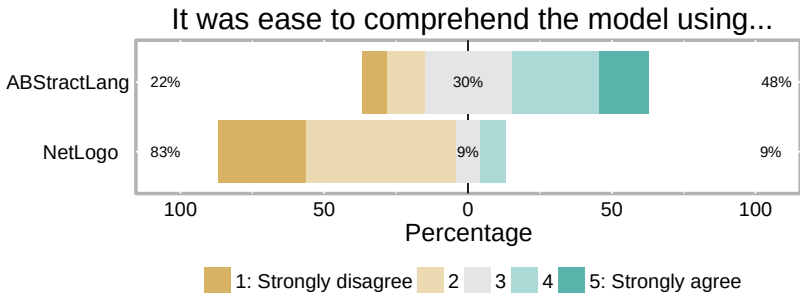
Although the overall *score* of groups using ABStractLang is greater than those using NetLogo, a Kruskal-Wallis test revealed no significant difference among groups ($\chi^2 = 7.565$, $p = 0.0559$). Despite this not statistically significant for the overall score, we observed that the higher score achieved by using ABStractLang is mainly due to questions regarding parameters identification (for Haiti) and parameters initialization (for TELL ME) and regarding entities and their attribute types (for Haiti) and entities creation (for TELL ME). This indicates that, at least for some agent-based simulation aspects, ABStractLang brings advantages with respect to correct comprehension.

Analyzing the mistakes made by participants, we noticed that some participants misunderstood some concepts of ABStractLang, causing them to answer many questions incorrectly. For instance, some did not understand that a spatial unit related to the spatial abstraction is an entity. This caused them to provide wrong answers to questions associated with entities, specifically those asking for the existing entities (Q05), their attributes (Q07) and how entities are created and situated (Q10 and Q11, respectively). Considering that these participants were not trained for interpreting ABStractLang elements (they received only a reference card), but they had previous experience with programming (all above the intermediate level), the familiarity with source code that describes their elements and types in a structured way might have helped understanding simulations using NetLogo. A training session on ABStractLang can potentially reduce misunderstanding, increasing its score, which is already higher, though not significant.

[2]Throughout this thesis, the Kruskal-Wallis test is used to compare more than two groups whenever the normality of data and homogeneity of variance assumptions do not hold. If these assumptions hold, then the ANOVA test is used. To compare only two groups, the Wilcoxon test is used.

Figure 5.3 shows the results regarding the perceived ease of comprehension considering 25 participants (one participant did not fill the post-test form). These results express participants level of agreement, in a 5-point Likert scale, to the following statement: "*It was easy to comprehend the agent model using {ABStractLang, NetLogo}*". It can be seen that participants agreed it is easy to comprehend agent-based simulation using ABStractLang (48% agree or strongly agree, 30% neutral), but disagreed that it is the case using NetLogo (83% disagree or strongly disagree, 9% neutral). A Wilcoxon Signed-ranks test indicates that the difference between ABStractLang ($M = 3.36$; $SD = 1.15$) and NetLogo ($M = 1.88$; $SD = 0.88$) is statistically significant ($W = 196.5$, $p < 0.01$).

Figure 5.3: Perceived Ease of Comprehension



Results of the user study give evidence that ABStractLang decreases the time to comprehend simulations having complex entities and many model parameters. In these cases, the more expressive notation adopted for these elements led to a more effective comprehension of their specification, creation, and initialization. For simple simulations, the comprehension time using ABStractLang seems to be about the same in comparison with NetLogo, at least when participants have prior programming expertise. This suggested the potential of using ABStractLang to comprehend simulations in less time. Moreover, based on the subjective opinion of participants, we observed that ABStractLang may also require less cognitive effort to understand simulations. Regarding comprehension, given that there is no significant difference among the overall scores, it was not possible to claim that ABStractLang increases the correct comprehension of implemented simulations (although this is not the case for specific simulation aspects). However, such a conclusion is valid under the particular setup of the experiment: participants that did not receive extensive training in ABStractLang but have experience with programming (which favors NetLogo).

5.1.4 Threats to Validity

This section discusses threats to study validity. An internal threat is that participants could have become tired while taking part of our study, performing worse while evaluating the second simulation. To address this, groups have distinct treatment orders, as previously shown in Table 5.2. This also addresses carry-over effects, given that both simulation model and technique were different in each development session.

An external threat is that our study was conducted only with computer scientists, some of them with basic knowledge in NetLogo. As said previously, this favors NetLogo and, therefore, differences would be even larger if this influenced the results.

5.2 Empirical Evaluation of Development Effort

In this section, we present an evaluation of the effectiveness of MDD4ABMS and its ABStractLang language to develop simulations in the adaptive traffic signal control application domain. The evaluation was conducted when aspects from this domain were abstracted into our metamodel. It consists of an empirical study in which a software engineering metric was used to assess the productivity of using MDD4ABMS to develop agent-based simulations.

The goal of the study is to assess the productivity gains promoted by MDD4ABMS. The metric we selected is the *effort* to develop agent-based simulation. In many software cost estimation models, the effort is a function of the size of the system: the smaller the size, the lower the effort required to develop it. Consolidated cost estimation methods such as Function Points (ALBRECHT, 1979), COCOMO 2.0 (BOEHM et al., 1995), SEER-SEM (JENSEN, 1983; Galorath Inc., 2017), and SLIM (PUTNAM; MYERS, 1991) rely on size metrics to estimate the required effort as person-months or calendar months. Other estimation models focused on web applications have considered size metrics as input (ABRAHÃO; GÓMEZ; INSFRAN, 2010). Indeed, previous work showed that system size is the most significant factor that affects development effort, quality and construction time (AGRAWAL; CHARI, 2007).

Therefore, our evaluation considers the size of simulation models to measure the effort required to produce them. More specifically, we use Lines of Code (LOC), which is frequently used as a software size metric (BOEHM et al., 1995; JENSEN, 1983; Galorath Inc., 2017; PUTNAM; MYERS, 1991). In fact, LOC is a metric that has been

used to evaluate and compare design and development implementation effort in MDD approaches (OSIS; ASNINA, 2010). This is also the case in MAS (CHALLENGER; KARDAS; TEKINERDOGAN, 2015). ABStractLang adopts a graphical representation for models. Thus, for a fair comparison, we use the Atomic Model Element (AME) metric (BETTIN, 2002) to measure the design development effort of ABStractLang simulation models. An AME is a visual modeling element that is equivalent to a LOC.

5.2.1 Procedure

To evaluate the effectiveness of our MDD4ABMS approach for modeling adaptive strategies recurrently used in traffic signal control simulations (adaptation, learning, and state machines), we selected one existing simulation for each of them. As discussed in the background chapter, existing MDD alternatives cover only conceptual simulation aspects and, consequently, existing simulations are usually implemented using agent-based simulation platforms or general purpose programming languages to have a runnable simulation. We thus compared MDD4ABMS to alternatives with which a runnable simulation can be developed. As we adopted a software size metric, availability of source code is crucial for a fair comparison and thus was adopted as the criteria for selecting the simulations of our study. Whenever available, simulations developed with NetLogo were selected, given that our MDD4ABMS approach generates code for this platform.

Considering that a state machine represents a fixed decision policy, we selected a simulation of fixed traffic signal plans (WILENSKY, 2003) for evaluating the effectiveness of modeling them as state machine capabilities. The simulation of self-organizing traffic lights (GERSHENSON, 2005) was selected to evaluate the *adaptation* capability. Both simulations are available for NetLogo.[3][4] Finally, to evaluate the effectiveness of using learning for traffic signal control, we used the simulation of traffic signal control agents with reinforcement learning (OLIVEIRA; BAZZAN, 2009). This simulation is available for the ITSUMO simulation platform (SILVA et al., 2005).

The LOC metric was collected from the available source code of selected simulations. When using LOC as input for effort estimation, a distinction between manually produced and automatically generated lines of code should be made. The former requires human effort for being produced, while the latter is produced by code generators.

[3] <http://ccl.northwestern.edu/netlogo/models/TrafficGrid>
[4] <http://turing.iimas.unam.mx/~cgg/sos/SOTL/SOTL.html>

Given that our focus is on traffic signal controller agents, our study considered only the code blocks dedicated to implement them and their associated behaviors. Code related to vehicle agents and the traffic network (simulated environment), as well as comments and code block delimiters, were not considered in this evaluation. To collect the number of AME in models created by using ABStractLang, we adapted the generous estimation rule proposed for UML class diagrams (BETTIN, 2002). The following elements were counted as AMEs: model parameters; entity or agent boxes; attributes and their corresponding initialization and update; agent creation strategy and each of its parameters; agent capabilities, their options, and parameters; actuator states and groups; and connections of agent capabilities.

5.2.2 Results and Discussion

Obtained results are shown in Table 5.4, separated by decision capability. Columns indicate: (i) AME: the number of specified atomic model elements; (ii) MLOC: the number of manually written lines of code; (iii) Development Effort: the sum of AMEs and MLOC, which is the (design and implementation) development effort of the manually created portion of the simulation model; and (iv) GLOC: the number of lines of code generated from AMEs. That table shows that the ABStractLang language requires less development effort to produce simulation models for all the three simulations. The workload of developers was reduced by 60.00%, 81.93%, and 85.21%, respectively. By using ABStractLang, all the human effort is employed to create diagram elements only (AME), from which the simulation source code is automatically generated. Consequently, no MLOC is required for MDD4ABMS. By using NetLogo, in addition to MLOC, one AME was used to specify model parameters via graphical input components in each simulation, producing some GLOC. On the other hand, for the simulation developed with ITSUMO, all the human effort is employed to create MLOC because no graphical input component was used in the existing simulation. We recall that results show in Table 5.4 considers only model and code elements related to traffic signal controller agents, which corresponds to approximately 50% of the complete model/code required to run a simulation. Vehicle and visualization-related source code corresponds to the remaining 50%. These aspects were not considered in this evaluation and therefore in order to run each simulation their source code were reused from existing implementations.

It is also possible to observe that the total amount of lines of code (MLOC +

Table 5.4: Empirical Evaluation of Development Effort: Size Comparison

Simulation / Technique		AME	MLOC	Development Effort*	GLOC
State Machine (Fixed Plan)/					
	NetLogo	1	34	35	1
	MDD4ABMS	14	0	**14**	116
Adaptation (Self-organizing Traffic Lights)/					
	NetLogo	1	82	83	1
	MDD4ABMS	15	0	**15**	103
Learning (Reinforcement Learning)/					
	ITSUMO	0	257	257	0
	MDD4ABMS	38	0	**38**	298

*Development Effort = AME + MLOC

GLOC) produced by the MDD4ABMS approach for traffic signal controller agents and their behavior is greater than the total amount of lines of code in the existing implementation, for all simulations. Given that the MDD4ABMS metamodel considers domain independent abstractions (e.g., state machines and other decision capabilities), generated code contains additional statements to implement these abstractions. Existing implementations, in turn, are built upon application-specific abstractions that may not generalize to other domains. Nevertheless, GLOC is not considered as evidence of development effort, because no human effort is required to produce them.

The presented results give evidence that our MDD4ABMS approach is effective to reduce the effort required for developing agent-based simulations. The effort to design models using ABStractLang is lower than the effort to implement code for NetLogo or ITSUMO. Given that an AME is equivalent to an MLOC, the sum of AMEs and MLOC produced by using MDD4ABMS is always lower than by using the simulation platforms. Although the number of AMEs produced by using ABStractLang is the highest, it is lower than the number of MLOC produced by using the simulation platforms, leading to a reduction in the combined design and implementation effort. Furthermore, the rule adopted for counting AMEs led to a fine-grained and platform independent evaluation. Therefore, a few elements of ABStractLang were counted as AMEs, but their size is not equivalent to, but lower than, a line of code (e.g., parameters of a learning technique). As a consequence, the evaluation favored implemented simulations, giving us stronger confidence that MDD4ABMS reduces the human effort required to develop agent-based simulations in the traffic signal control domain.

5.2.3 Threats to Validity

An internal threat to study validity is that the AME metric is based on an estimation rule to classify elements of ABStractLang as AMEs. As said, we adopted a generous estimation rule (BETTIN, 2002). In spite of a few elements of ABStractLang were counted as AMEs, their size is indeed lower than a line of code. Therefore, differences would have been even larger if a more strict estimation had been adopted.

An external threat to validity is that we considered one agent type (traffic signal controller), which may raise issues with respect to the scalability of MDD4ABMS and its modeling language. In fact, scalability can be a problem in languages with graphical notation. We mitigated this problem in the ABStractLang language by introducing the notion of *concerns*, as explained in Chapter 3. In this empirical evaluation, we focused exclusively on adaptive traffic signal control simulations in which agents in charge of managing traffic signal controllers are the main aspect considered. Therefore, we deal exclusively with the concern of traffic signal controller agents. Additional aspects, such as microscopic vehicle models, would be modeled as additional concerns within their own diagrams (e.g., vehicles concern), and thus would not compromise the scalability of our modeling language and the validity of our results.

The modeling of one simulation for each decision capability is another external threat to validity. Additional simulations were enumerated when the evaluation was planned, either related to learning (ABDOOS; MOZAYANI; BAZZAN, 2011; MANNION; DUGGAN; HOWLEY, 2016; ROSSETTI et al., 2002) or adaptation (GERSHENSON; ROSENBLUETH, 2012b; GERSHENSON; ROSENBLUETH, 2012a). When examining how these simulations deal with strategies for fixed traffic signal plans, adaptation, and learning, we noticed that they differ not by the techniques adopted, but by its setup. For example, both Mannion, Duggan and Howley (2016) and Oliveira and Bazzan (2009) use Q-Learning as reinforcement learning technique but adopt different state representations and reward functions. Therefore, this gives evidence that the selected simulations represent elements that are recurrent in agent-based adaptive traffic signal control simulations that adopt one of the three adaptive strategies considered in this work, thus mitigating the threat regarding our selected simulations.

5.3 User Study: Simulation Development with MDD4ABMS

In the evaluations described so far, elements of our MDD4ABMS approach were considered separately. In this section, we describe a study with humans that considered our MDD4ABMS approach as a whole. Following the GQM template, the goal of the study is: *to assess the benefits of using MDD4ABMS to develop agent-based simulations, evaluate the MDD4ABMS approach in comparison with NetLogo from the perspective of the researchers as it is used by mainstream software developers with little ABMS expertise to develop agent-based simulations in a multi-project study*. The study considers participants with little or no expertise in ABMS so as to be consistent with the hypothesis considered in this thesis, of that an MDD approach improves productivity in agent-based simulation development for developers with this level of expertise. Additionally, we conducted the evaluation as a multi-project study to avoid domain biases.

To achieve the study goal, we derived three research questions, stated as follows.

RQ1. Does MDD4ABMS improve *design quality* of agent-based simulations?

RQ2. Does MDD4ABMS decrease the *effort* required to develop agent-based simulations?

RQ3. How do developers evaluate MDD4ABMs with respect to *usability, reliability, productivity*, and *expressiveness*?

One of the goals of MDD is to enhance software quality (STAHL et al., 2006b). Therefore, RQ1 aims at verifying how MDD4ABMS affects the design quality of agent-based simulations. Research question RQ2 is motivated by the claims of ABMS researchers regarding the demand for alternatives that ease the development of agent-based simulations and increase development productivity. As previously discussed, despite the potential of ABMS, one who wants to build a simulation may have motivation impaired if the required effort is high and therefore beyond his abilities or budget. Previous work on the use of MDD in many domains showed that it is a productive and effortless alternative for software development. Therefore, RQ2 intends to verify whether this is also observed in the ABMS paradigm. Finally, RQ3 is focused on evaluating MDD4ABMS with respect to the user experience. Existing methodologies for evaluating MDD approaches and their DSLs point out that qualitative aspects such as those enumerated in RQ3 influence both the success and adoption of such approaches (CHALLENGER; KARDAS; TEKINERDOGAN, 2015; KAHRAMAN; BILGEN, 2015).

We selected the following metrics to answer these research questions.

M1. Number of *correct features* in the simulation. (RQ1)

M2. Number of *incomplete features* in the simulation. (RQ1)

M3. Number of *features with syntactical errors* in the simulation. (RQ1)

M4. Number of *inconsistent features* in the simulation. (RQ1)

M5. *Time* spent to develop agent-based simulations. (RQ2)

Metrics M1 to M4 are related to the design quality of simulations. As Hoffert, Schmidt and Gokhale (2011), we conceive design quality as the number of defects in a simulation. Defects are accounted at the same granularity level of the simulation features. Features correctly developed are considered as *correct features* (M1). The level at with a feature is correct is determined by inspecting the simulation model (MDD4ABMS) or implementation (NetLogo). To accomplish a particular feature, a set of elements must be present in the simulation. Therefore, a feature is considered entirely correct if all its expected elements are specified by the participant. Three types of defects are considered in the study. Features requested but not fully developed (i.e., some element expected for that feature is missing) are considered as defects of type *incomplete features* (M2). Features developed with compilation errors are considered as defects of type *syntactical error* (M3). Finally, features developed without compilation errors but that do not produce the expected simulation results are considered as defects of type *inconsistent features* (M4)—i.e., the participant endued the agent with a learning capability but specified wrong learning parameters.

Metric M5 measures the time it took participants to develop an agent-based simulation and to verify whether it produces the expected results. Existing evaluations of MDD approaches have already considered time as a measure for development effort (HOFFERT; SCHMIDT; GOKHALE, 2011).

To answer research question RQ3, we adopted the Framework for Qualitative Assessment of DSLs (FQAD) by Kahraman and Bilgen (2015). FQAD enumerates a set of qualitative aspects relevant to the success of DSLs and provides an instrument (questionnaire) for assessing them. Though FQAD was conceived for evaluating DSLs, in this study it reflects the assessment of MDD4ABMS as a whole.

5.3.1 Procedure

The user study was conducted as a final assignment of an agent-based simulation mini-course. The mini-course had a total of 15 working hours, comprising a hands-on tutorial[5] and an experiment session (12 and 3 hours respectively). In the tutorial session, participants were introduced to the ABMS paradigm as well as to the MDD4ABMS approach. Much of the tutorial session was dedicated to practice the development of agent-based simulations in the domain of spread of disease and adaptive traffic signal control so as to participants learn the languages and tools used in the experiment (ABStractLang from MDD4ABMS, and NetLogo). Providing mini-courses instead of just experiment sessions were the way we found to increase the number of volunteer participants.

In the experiment session, two agent-based simulations had to be developed in steps, and in each of which participants were asked to develop a particular simulation feature. To avoid domain biases, one simulation explores the domain of spread of diseases, and the other explores the domain of adaptive traffic signal control (hereafter referred to as *disease* and *traffic*, respectively). Details on these simulations are described next.

The spread of disease simulation aims at reproducing the perpetuation of a disease in a population of agents (NATHANSON, 2005). Two agents are considered in this simulation: natives and immigrants. Both are subject to the same disease, specified via the SIR compartmental model. The way the disease affects an agent depends on its type, and therefore the SIR model parameters (e.g., transmission and recovery rates) must be set accordingly. The features developed by participants, and the associated development task, are presented next. Details on the number of agents and the SIR model parameters are presented in Appendix E.

DF1. *Environment*: creation of a grid with fixed size.

DF2. *Native agent*: creation of the native agents, which are positioned in the environment according to a GIS file and randomly move around the environment.

DF3. *Disease in natives*: incorporation of the SIR compartmental model into natives, following a specification that gives the transmission, recovery, and mortality rates, as well the immunity duration and how the disease is introduced in the population of natives. Participants also had to specify outputs to observe the number of susceptible, infected, and recovery natives during the simulation. Finally, agents are colored according to their current compartment (provided as

[5] Available, in Portuguese, at <https://sites.google.com/view/simulacoescomagentes/>

an external agent capability to be added to the simulation).

DF4. *Immigrant agent*: creation of immigrant agents, which moves exactly as natives, with a fixed population.

DF5. *Disease in immigrants*: the SIR compartmental model is incorporated into immigrants (with specified parameters), as well as output and coloring similar to natives. Now, the disease transmission can also take place between natives and immigrants. Participants also had to specify outputs to observe immigrant agents, as well as to add external agent capability for coloring immigrants.

In the adaptive traffic signal control simulation, the goal is to demonstrate and evaluate how traffic signal control agents using reinforcement learning can learn a policy for selecting a traffic signal plan that minimizes the number of stopped vehicles. Two agents are considered: traffic signal controllers (TSCs) and vehicles. TSCs are provided with three plans, each of which gives priority to vehicles moving in a particular direction. By using a reinforcement learning technique that considers these plans in addition to learning states and a reward function, TSCs can identify which plan is best suitable to handle a particular traffic demand. Features developed by participants are presented next. Details on the number of agents and reinforcement learning parameters are presented in Appendix F.

TF1. *Environment*: creation of a graph (traffic network) from an OSM file.

TF2. *Vehicle agent*: inclusion of vehicle agents, following a given specification.

TF3. *TSC agent*: creation of TSC agents at each intersection node of the traffic network.

TF4. *Traffic signal plans*: specification of three traffic signal plans to TSC agents, and set up of them to control the flow of vehicles at intersections.

TF5. *Reinforcement learning*: addition of the reinforcement learning technique (more specifically the *Q-learning* algorithm) to TSC agents, and set up of learning states, actions, reward function, and other learning parameters. The reward function punishes TSCs according to the sum of the queue length at all the incoming lanes (the more the number of stopped vehicles, the higher is the punishment; in the best case, with no stopped vehicles, the punishment value is zero). Participants also had to specify an output to observe the overall number of stopped vehicles during the simulation.

Once again, we chose the NetLogo platform as the baseline for evaluating the

benefits provided by MDD4ABMS. The NetLogo programming language provides high-level statements for ordinary operations (e.g., moving agents and finding elements located within a given radius) required in the considered simulations. Additionally, code blocks of agent behaviors found in existing NetLogo models can be extended and reused in simulations, fostering the adoption of this platform. Indeed, this was the case for the traffic simulation in our study, in which source code of both the vehicle agent and the *Q-learning* algorithm were reused from existing models (RODRÍGUEZ-HERNÁNDEZ; BURGUILLO-RIAL, 2016; ROOP, 2006). In these cases, participants that developed simulations with NetLogo received ready-to-use libraries with these features and had to incorporate them into simulations. Finally, given that MDD4ABMS generates NetLogo source code, participants have not had to learn how to use other simulation platforms. However, elementary programming skills are expected from participants because teaching programming logic is out of the scope of the user study.

To study the two independent variables considered in our study—simulation domain (disease or traffic) and development technique (MDD4ABMS or NetLogo)—four treatment groups were adopted in the experiment session of the mini-course. Each participant was randomly assigned to one of these groups. In each group, the time it took participants to develop simulations (metric M5) is collected through two consecutive development sessions, in which the treatment order was changed. A survey tool (Lime Survey) was used to collect the time it took participants to develop each simulation feature.

Metrics M1 to M4 are calculated from the simulations finished by participants. Each simulation was inspected to verify whether features were correctly developed. A feature is considered correct if all the expected constructs for that feature were specified by the participant. The expected constructs depend on the technique used to develop the simulation. For example, to specify a disease by using MDD4ABMS, a disease model capability element correctly configured is expected. By using NetLogo, it is expected a set of code statements that implement all the disease-related processes (transmission, recovery, mortality, and introduction).

To answer RQ3, by the end of the two development sessions participants were asked to fill out the questionnaire provided by FQAD (KAHRAMAN; BILGEN, 2015). In this questionnaire, participants agree or disagree on the presence of qualitative aspects using a 5-point Likert scale, for both MDD4ABMS and NetLogo. The following qualitative aspects were considered in this study: (i) *usability*: the degree at which the approach

can be used by participants to achieve their goals; (ii) *reliability*: whether the approach aids producing simulations free of errors and mistakes; (iii) *productivity*: the degree at which the approach promotes productivity; and (iv) *expressiveness*: the degree at which it eases the development of simulations by providing elements at the right abstraction level. Additionally, two open-ended questions asking participants to provide the most positive and most negative aspects of both techniques were included in the questionnaire, as suggested by Lund (2001).

5.3.2 Participants

Considering that having got programming skills is a prerequisite to attend the mini-course, we targeted graduate and undergraduate students of Computer Science and related courses. Three mini-course editions took place between May 5 and July 20, 2018, in two distinct sites. The first site was the Universidade Regional de Blumenau (FURB), in Blumenau/SC, and the second was the Instituto de Informática of the Universidade Federal do Rio Grande do Sul (UFRGS), in Porto Alegre/RS.

Table 5.5 presents the number of participants in each site/edition of the mini-course. In the UFRGS/1 edition, six participants dropped out from the tutorial session: 25 participants started, but only 19 finished the tutorial. In a follow-up interview with these drop-outs, respondents pointed out that scheduling or health issues prevented them from finishing the mini-course. There were drop-outs at the FURB/1 edition as well. Although all participants finished the tutorial, six did not participate in the experiment session due to scheduling issues.

Table 5.5: Participants of the Mini-course Editions

		Tutorial		**Experiment**		
Site/Edition	**Date**	**Started**	**Finished**	**Started**	**Finished**	**Considered**
FURB/1	May 5 and 12	19	19	13	13	12
UFRGS/1	July 10 to 13	25	19	19	17	14
FURB/2	July 16 to 20	8	8	8	8	5
	Total	**52**	**46**	**40**	**38**	**31**

With respect to the experiment session, drop-outs were registered only in the UFRGS/1 edition, in which two participants started the experiment session, but instead of developing the simulations, they started to work on some other activity (e.g., preparation for final exams). These two participants were kindly asked to drop-out from the

experiment session. Given that these drop-outs were caused neither by difficulties with the techniques nor with the domains, they are not an indication of issues with a particular treatment. The last column in Table 5.5 shows the number of participants whose simulations were indeed considered for evaluation. From the 38 participants that finished the experiment, 7 were discarded from the study due to the reasons described next.

- *Delayed start*. Two participants in the UFRGS/1 edition arrived late at the experiment session. They were authorized to participate in the session to fulfill the credit hours and then receive the certificate of participation. However, these participants were removed from the study because they missed the explanation of the experimental procedure.
- *Technical issues*. Issues with the online survey tool were registered by one participant in the FURB/1 and UFRGS/1 editions and by two in the FURB/2 edition. In that cases, the survey tool either did not record the time they have taken to develop simulation features (session timeout issues) or did not show the instructions for developing a particular feature (page loading issues). A runtime error (memory leak) in the ABStractme tool compromised the model created by a participant in the FURB/2 edition.[6] The error affected the time recorded for the participant, and therefore this participant was removed from the study.

Table 5.6 presents demographic characteristics of the thirty-one participants considered. Participants were mostly male, undergraduate students in Computer Science. Before the mini-course, participants were asked to quantify (9-point Likert scale) their expertise in the following topics related to this study: programming; agent-based modeling and simulation; traffic control and traffic lights; epidemiology, in particular, epidemics (e.g., compartmental models); reinforcement learning techniques and algorithms; the NetLogo platform and its programming language; and UML class diagrams.

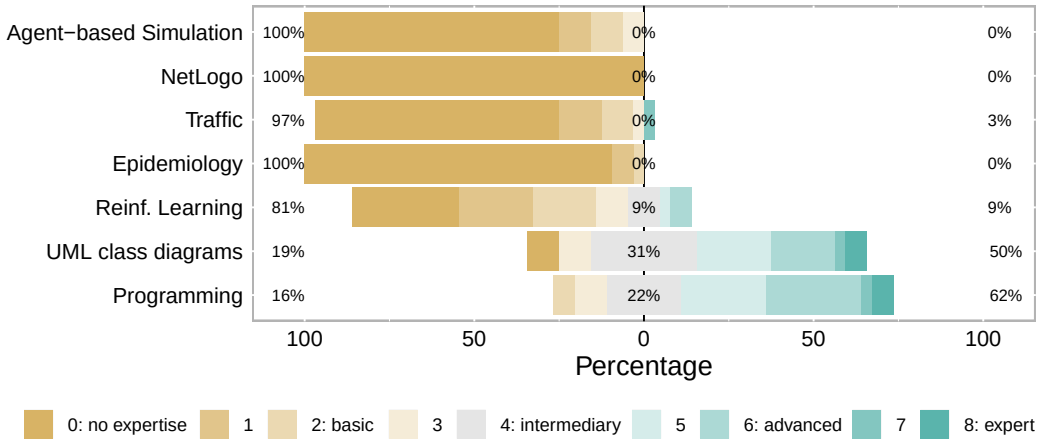
Figure 5.4 presents expertise reported by participants. It can be seen that almost all participants reported no expertise in agent-based simulations (only a few reported expertise around the basic level in this topic). No participant reported expertise in NetLogo. Therefore, the sample of participants conforms with the goal of this study, which is to assess the benefits of MDD4ABMS for people with little expertise in ABMS. Participants also reported little expertise in topics such as traffic, epidemiology, and reinforcement learning. Consequently, no participant needed to be excluded due to prior knowledge of

[6]The error has already been fixed in the current release of the ABStracme tool.

Table 5.6: Demographic Characteristics of Participants

Characteristics		**N**	**%**
Gender	Female	3	9.68
	Male	28	90.32
Age	15 – 20	9	29.032
	21 – 25	11	35.483
	26 – 30	7	22.580
	31 – 35	3	9.677
	36 – 40	1	3.225
Area	Computer Science	29	93.55
	Computer Engineering	1	3.23
	Information Systems	1	3.23
Education	Undergraduated	25	80.65
	Graduated	2	6.45
	Specialization student	2	6.45
	Master student	1	3.23
	PhD student	1	3.23

Figure 5.4: Expertise of Participants



the technologies under study. 50% of participants reported expertise in UML class diagrams above the intermediary level. Participants reported considerable expertise in programming, as expected from Computer Science students. To better characterize programming skills, Table 5.7 presents the number of years of programming expertise reported by participants. With respect to programming, almost half of the participants reported 2–5 years of expertise. With respect to programming as professionals (e.g., working in industry), in turn, there are participants with no such expertise, as well as participants with 5 or more years of professional programming.

As detailed earlier, four treatment groups were considered in the study. Participants were randomly assigned to one of these four groups and developed the simulations in the sequence and with the technique specified for that group. Table 5.8 shows the treat-

Table 5.7: Programming Expertise of Participants, in Years

Characteristics		**N**	**%**
Programming Expertise (in years)	No expertise	0	0.00
	< 1	0	0.00
	1 – 2	5	13.13
	2 – 5	13	41.94
	5 – 10	11	35.48
	> 10	2	6.45
Professional Programming Expertise (in years)	No expertise	9	29.03
	< 1	3	9.68
	1 – 2	6	19.35
	2 – 5	8	25.81
	5 – 10	4	12.90
	> 10	1	3.23

ment conditions and the number of participants of each group after drop-outs previously described. Under this configuration, a total of 62 simulations were developed. For each combination of domain and technique, the number of developed simulation is the following: *Disease-MDD4ABMS*: 15; *Disease-NetLogo*: 16; *Traffic-MDD4ABMS*: 16; and *Traffic-NetLogo*: 15. Results of the experiment are presented and discussed next.

Table 5.8: Number of Participants per Treatment Group

		Development Session 1		**Development Session 2**	
Group	**N**	**Domain**	**Technique**	**Domain**	**Technique**
A	9	Disease	MDD4ABMS	Traffic	NetLogo
B	8	Traffic	MDD4ABMS	Disease	NetLogo
C	8	Disease	NetLogo	Traffic	MDD4ABMS
D	6	Traffic	NetLogo	Disease	MDD4ABS

5.3.3 Results and Discussion

The results obtained are next presented and discussed by research question. We first present results regarding design quality (RQ1), followed by results regarding development effort (RQ2). Lastly, we present the subjective evaluation of MDD4ABMS with respect to the user experience (RQ3).

5.3.3.1 Design Quality (RQ1)

The number of features correctly developed by participants in each simulation (metric M1) is summarized in Figure 5.5 and detailed in Table 5.9, which shows the mean

Figure 5.5: Number of Correct Features

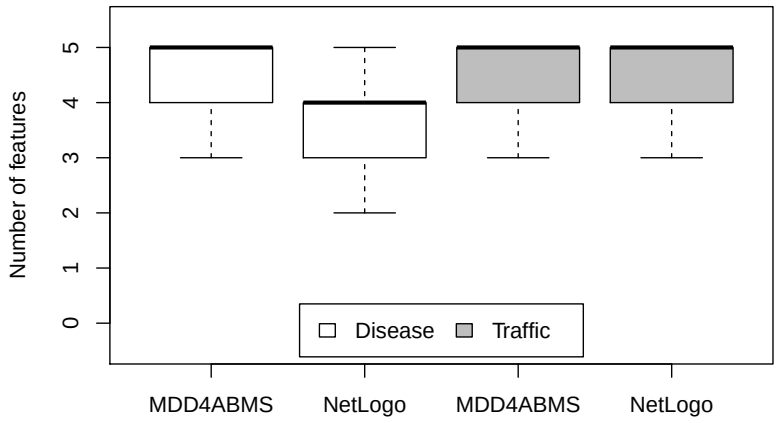


Table 5.9: Details on the Number of Correct Features

Domain / Technique	**Mean (SD)**	**Number of simulations with N correct features** N=1	N=2	N=3	N=4	N=5
Disease /						
MDD4ABMS	4.53 (0.64)	0 (0.00%)	0 (0.00%)	1 (6.66%)	5 (33.33%)	**9 (60.00%)**
NetLogo	3.69 (0.79)	0 (0.00%)	1 (6.25%)	5 (31.25%)	**8 (50.00%)**	2 (12.50%)
Traffic /						
MDD4ABMS	4.56 (0.63)	0 (0.00%)	0 (0.00%)	1 (6.25%)	5 (31.25%)	**10 (62.50%)**
NetLogo	4.53 (0.64)	0 (0.00%)	0 (0.00%)	1 (6.66%)	5 (33.33%)	**9 (60.00%)**

and standard deviation, in addition to the number of simulations according to the number of correct features. It can be seen that in the disease domain the average number of correct features developed by using MDD4ABMS (4.53) is 25.14% higher than by using NetLogo (3.69). In this domain, 60.00% of the simulations developed by using MDD4ABMS (9 out of 15) were entirely correct (i.e., all 5 features correctly developed), while by using NetLogo most of the simulations presented either 3 (31.25%) or 4 (50.00%) correct features, and only 12.50% (2 out 16) were entirely correct. In the traffic domain, in turn, results obtained for MDD4ABMS and NetLogo are similar. These observations are confirmed by the significant difference among the groups that was revealed by a Kruskal-Wallis test ($\chi^2 = 14.04$, $p < 0.01$), followed by a post hoc Dunn's test with Holm correction that showed a significant difference between MDD4ABMS and NetLogo in the disease domain.

Features that were incorrectly developed were further analyzed and categorized as: *incomplete* (metric M2); *syntactical error* (metric M3); and *inconsistent* (metric M4). Table 5.10 shows the number of simulations in which these defects were found. There are incomplete features in both domains. In the disease domain, by using NetLogo, the number of simulations with incomplete features (31.25%) is higher than by using MDD4ABMS (6.67%). By using NetLogo, the number of incomplete features per simulation is 1.20 on

Table 5.10: Number of Simulations and Features by Defect Type

Domain	Technique	Nb. of Simulations	Nb. of features with error per simulation Mean	SD	Min	Max
Incomplete:						
Disease	MDD4ABMS	1 (6.67%)	1.00	-	1	1
	NetLogo	5 (31.25%)	1.20	0.45	1	2
Traffic	MDD4ABMS	3 (18.75%)	1.33	0.58	1	2
	NetLogo	3 (20.00%)	1.00	0.00	1	1
Syntactical Error:						
Disease	MDD4ABMS	0 (0.00%)	-	-	-	-
	NetLogo	1 (6.25%)	1.00	-	1	1
Traffic	MDD4ABMS	0 (0.00%)	-	-	-	-
	NetLogo	1 (6.67%)	1.00	-	1	1
Inconsistent:						
Disease	MDD4ABMS	6 (40.00%)	1.17	0.41	1	2
	NetLogo	13 (81.25%)	1.38	0.51	1	2
Traffic	MDD4ABMS	4 (25.00%)	1.00	0.00	1	1
	NetLogo	4 (26.67%)	1.25	0.50	1	2

average, which means that there are simulations with more than one incomplete feature. In such simulations, incompleteness is found in features DF3 and DF5, caused by missing code statements for initializing and managing the disease progression, and for using the provided agent coloring routines. By using MDD4ABMS, only feature DF5 presents incompleteness, caused by a missing element in the immunity duration specification. In the traffic domain, the number of simulations with incomplete features is the same for each technique (percentages are not the same due to different number of participants in each group, as shown in Table 5.8). For both techniques, there are cases in which the simulation output is missing in feature TF5. By using MDD4ABMS, there is a simulation in which a particular traffic signal plan was not added (TF4), and another with a missing connection between one plan state machine and the reinforcement learning agent capability (TF5), causing the learning to consider only two out of the three traffic signal plans. By using NetLogo, there are two simulations with missing code statements to activate the reinforcement learning technique (TF5).

There are cases of features with syntactical errors only in simulations developed by using NetLogo. Errors are due to syntax issues in assignment operations and the use of agent attributes (e.g., wrong attribute names).

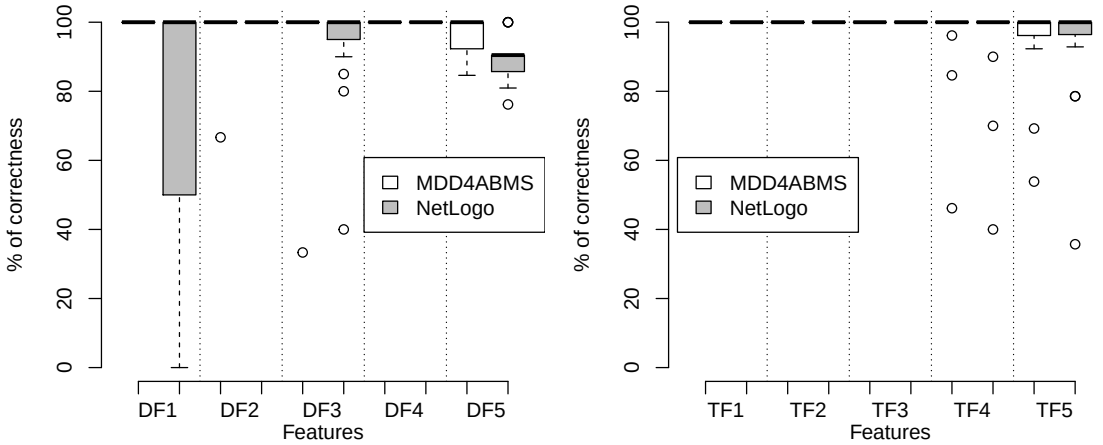
Finally, there are cases of inconsistent features in both domains. In the disease domain, the number of simulations with inconsistent features developed by using NetLogo

(81.25%) is twice as by using MDD4ABMS (40.00%). For both techniques, most of the inconsistencies occurred while developing features DF3 and DF5, in which participants had to specify the disease for the native and immigrant agents, respectively. The most common problem is the wrong specification of transmission rates for DF5. While by using MDD4ABMS this inconsistency is present in 3 out of the 6 simulations with inconsistent features, by using NetLogo it occurs in 11 out of the 13 simulations. Other less frequent inconsistencies are the following. By using MDD4ABMS: wrong mobility parameters, wrong infection duration or disease introduction, and wrong outputs. By using NetLogo: wrong environment size, and wrong infection and immunity durations. With respect to the traffic domain, the number of simulations with inconsistent features is the same. For both techniques, all inconsistencies occurred while developing features TF4 and TF5 (traffic signal plans and reinforcement learning, respectively). In TF4, one simulation had state machines for traffic signal plans incorrectly modeled by using MDD4ABMS, while by using NetLogo there are three simulations with inconsistent cycle or plan durations. In TF5, two simulations had output parameters incorrectly specified by using MDD4ABMS, while two simulations had wrong learning parameters by using NetLogo. This indicates that mixing programming logic with the specification of simulation parameters can possibly induce developers to make mistakes, even when the parameters are given.

So far, we discussed design quality from the point of view of the number of entirely correct features. Given that portions of features may have been correctly developed by participants, Figure 5.6 shows the percentage of correctness of features. These results are detailed in Table 5.11 and discussed next.

We first discuss results associated with the disease domain, shown in Figure 5.6(a).

Figure 5.6: Percentage of Correctness of Features



(a) Domain: Disease

(b) Domain: Traffic

Table 5.11: Details on Percentage of Correctness of Features

Feature	Technique	Percentage of Correctness							
		Disease				Traffic			
		Mean	SD	Min	Max	Mean	SD	Min	Max
1	MDD4ABMS	100.00	0.00	100.0	100.00	100.00	0.00	100.00	100.00
	NetLogo	75.00	44.42	0.00	100.00	100.00	0.00	100.00	100.00
2	MDD4ABMS	97.78	8.61	66.67	100.00	100.00	0.00	100.00	100.00
	NetLogo	100.00	0.00	100.0	100.00	100.00	0.00	100.00	100.00
3	MDD4ABMS	95.56	17.21	33.33	100.00	100.00	0.00	100.00	100.00
	NetLogo	93.44	15.57	40.00	100.00	100.00	0.00	100.00	100.00
4	MDD4ABMS	100.00	0.00	100.00	100.00	95.43	13.70	46.15	100.00
	NetLogo	100.00	0.00	100.00	100.00	93.33	16.76	40.00	100.00
5	MDD4ABMS	95.90	6.41	84.62	100.00	94.23	13.32	53.85	100.00
	NetLogo	89.88	6.48	76.19	100.00	92.38	17.39	35.71	100.00

In feature DF1 participants only had to specify a grid environment. Despite being an easy to develop feature, in some simulations, the percentage of correctness is below a hundred percent by using NetLogo—75.00% on average, with a minimum of 0.00% (four simulations). In feature DF2, the number of native agents and their locations was provided by a GIS file. Percentages of correctness are similar for this feature. In feature DF3, participants had to implement all the logic to spread the disease among agents and to recover or kill them after the infection duration by using NetLogo, leading them to make more mistakes when developing this feature. By using MDD4ABMS, participants only had to specify a disease capability and fill it with the disease parameters. In feature DF4, participants had to specify the immigrant agent and create a fixed population at random locations. As for feature DF2, percentages of correctness are similar. Finally, in feature DF5 participants had to subject the immigrant agent to the disease. Once again, by using NetLogo participants had to implement all the transmission and recovery logic, while by using MDD4ABMS participants were able to reuse the disease capability and had only to specify additional disease parameters related to the immigrant agent. A remarkable difference in the percentage of correctness is observed in this feature: 95.90% for MDD4ABMS, and 89.88% for NetLogo, on average.

With respect to the traffic simulation—Figure 5.6(b), results are as follows. In feature TF1, participants had to specify a graph for the environment and initialize if from a OSM file. In features TF2 and TF3, participants had to specify the vehicle and the traffic signal controller agent types, respectively. The percentage of correctness is 100.00% for these three features. In features TF4 and TF5, participants had to develop the traffic signal plans to control the flow of vehicles at intersections and to incorporate the reinforcement

learning technique, respectively. While these features demand writing many lines of code by using NetLogo, MDD4ABMS provides built-in agent capabilities for these elements. Although there were participants that made a few mistakes with both techniques, the average of the percentage of correctness of features developed by using MDD4ABMS is slightly greater than those developed by using NetLogo.

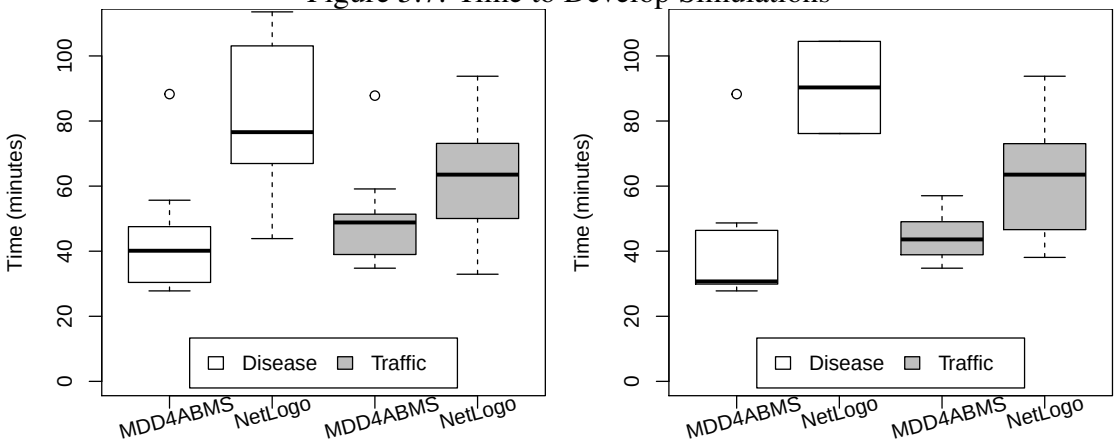
Findings: Design Quality (RQ1). The design quality of simulations developed by using MDD4ABM is at least as good as those developed by using NetLogo. In the particular case of the disease domain, design quality is superior considering the number of entirely correct features developed by using MDD4ABMS.

5.3.3.2 Development Effort (RQ2)

So far, we observed how correct the simulations developed by participants are. Now, we focus on the time it took participants to perform development tasks (metric M5). Results are summarized in Figure 5.7 and detailed in Table 5.12. Figure 5.7(a) shows the time it to develop finished simulations, including those with either incomplete or inconsistent features, as well as features with syntactical errors. Figure 5.7(b) shows the time it took participants to develop only the entirely correct simulations, i.e., those in which all the features are 100% correct. It can be seen that by using MDD4ABMS it took participants less time to develop simulations in both domains.

With respect to the time it took to develop finished simulations in the disease domain, by using MDD4ABMS it took participants 47.52% less time (42.27min, on average) than by using NetLogo (80.56min, on average). Similarly, in the traffic domain,

Figure 5.7: Time to Develop Simulations



(a) Finished Simulations (b) Entirely Correct Simulations

Table 5.12: Summary of Time (in Minutes) to Develop Simulations

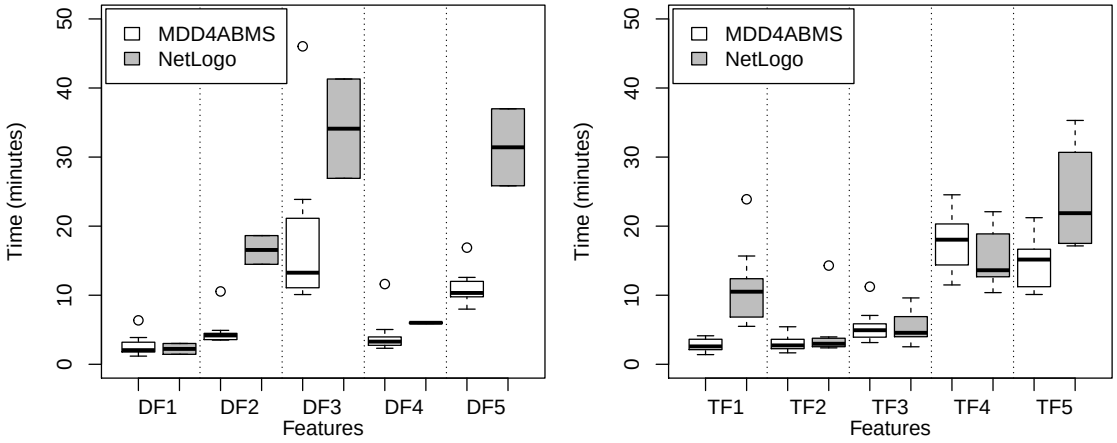
Domain	Technique	Mean	SD	Min	Max
Finished Simulations:					
Disease	MDD4ABMS	**42.27**	15.88	27.80	88.28
	NetLogo	80.56	20.56	43.87	113.54
Traffic	MDD4ABMS	**48.62**	12.90	34.78	87.81
	NetLogo	62.23	17.67	32.92	93.76
Entirely Correct Simulations:					
Disease	MDD4ABMS	**40.57**	19.48	27.80	88.28
	NetLogo	90.32	20.02	76.16	104.48
Traffic	MDD4ABMS	**44.06**	7.27	34.78	57.05
	NetLogo	61.47	17.98	38.09	93.76

by using MDD4ABMS it took 21.87% less time (48.62min, on average) than by using NetLogo (62.23min on average). A two-way ANOVA test showed a significant effect of *technique* on *time* at the $p < 0.05$ level [$F = 15.992$, $p < 0.001$]. A Tukey post hoc test revealed significant differences between MDD4ABMS and NetLogo in the disease domain ($p < 0.001$). With respect to the entirely correct simulations, the time it took to develop the disease simulation by using MDD4ABMS (40.57min, on average) is 55.08% less than by using NetLogo (90.32min, on average). By using MDD4ABMS to develop the traffic simulation it took participants 28.32% less time (44.06min on average) than by using NetLogo (61.47min, on average). A Kruskal-Wallis test revealed significant differences among the groups ($\chi^2 = 12.45$, $p < 0.01$), and a post hoc Dunn's test with Holm correction showed significant differences on time between MDD4ABMS and NetLogo in the disease domain.

Figure 5.8 summarizes the time it took participants to develop features, considering only the entirely correct simulations. These results are detailed in Table 5.13. It can be seen that by using MDD4ABMS participants took less time, on average, to develop most of the features. The more observable differences are in features that demand sophisticated constructs.

With respect to the disease simulation—Figure 5.8(a)—feature DF1 was quickly developed by using both techniques. However, as previously pointed out in the discussion regarding design quality (Section 5.3.3.1), by using NetLogo participants made a few mistakes when specifying the grid environment, while by using MDD4ABMS they did not. In feature DF2, by using NetLogo participants had to write a couple of statements to open the file, read its contents, and create the agents. By using MDD4ABMS, in turn, participants had to specify a creational strategy and refer it to the file, which took less

Figure 5.8: Time to Develop Features in Entirely Correct Simulations



(a) Disease (b) Traffic

Table 5.13: Details on Time to Develop Features in Entirely Correct Simulations

		Time (minutes) to Develop Features							
Fea-		**Disease**				**Traffic**			
ture	**Technique**	**Mean**	**SD**	**Min**	**Max**	**Mean**	**SD**	**Min**	**Max**
1	MDD4ABMS	2.73	1.59	1.18	6.36	2.83	0.96	1.40	4.13
	NetLogo	2.83	0.96	1.40	4.13	11.26	5.73	5.50	23.89
2	MDD4ABMS	4.79	2.21	3.50	10.55	2.97	1.14	1.66	5.43
	NetLogo	16.55	2.94	14.47	18.63	4.29	3.79	2.38	14.29
3	MDD4ABMS	17.90	11.62	10.10	46.03	5.50	2.32	3.14	11.23
	NetLogo	34.11	10.17	26.92	41.30	5.41	2.52	2.54	9.61
4	MDD4ABMS	4.19	2.92	2.32	11.61	17.79	3.95	11.49	24.54
	NetLogo	6.01	0.14	5.91	6.11	15.44	4.00	10.38	22.09
5	MDD4ABMS	10.96	2.67	7.99	16.89	14.97	3.70	10.10	21.22
	NetLogo	31.41	7.88	25.84	36.99	25.06	7.59	17.14	35.30

time on average. In feature DF3, by using NetLogo participants had to implement all the disease-related logic. By using MDD4ABMS, participants had to specify a disease capability, which took less time and led to a slightly higher percentage of correctness in comparison to NetLogo. In feature DF4, participants quickly developed the immigrant agent and the creation of a fixed population by using both techniques. Finally, in feature DF5 (subject the immigrant agent to the disease), participants were able to reuse the disease capability by using MDD4ABMS, which decreased the time it took to develop this feature in comparison to by using NetLogo.

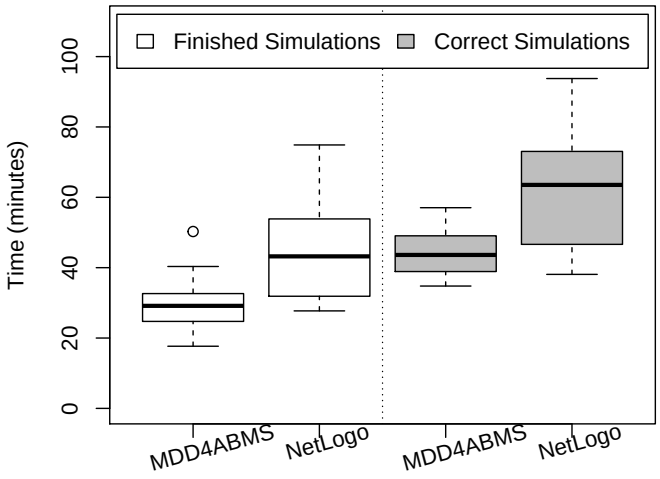
With respect to the traffic simulation—Figure 5.8(b)—by using MDD4ABMS, it took participants less time to develop features TF1 and TF5, and similar time for the remaining features. In feature TF1, by using NetLogo participants had to develop data types for graph nodes and links, and write statements to open and read the file. In contrast, MDD4ABMS provides a graph environment and participants had only to refer to the OSM

file, which took less time. In features TF2 and TF3, the time it took participants to develop the vehicle and the traffic signal controller agents are similar. Also in feature TF4 it took participants similar time to develop the traffic signal plans. Lastly, by using MDD4ABMS it took participants less time to develop feature TF5 given that MDD4ABMS provides a built-in learning capability.

For feature TF4, it took participants a modest higher time, on average, by using MDD4ABMS in comparison with NetLogo. To investigate this unexpected result, we performed a follow-up interview with the participants. Among the respondents, many pointed out that specifying traffic signal plans by using MDD4ABMS is not as straightforward as by using NetLogo. They mentioned difficulties in remembering how to create and configure flow control and state machine agent capabilities—the former is the means of controlling the flow of vehicles at intersections and the later the means of specifying traffic signal plans. Indeed, results previously presented regarding defects in features showed that they were mainly due to state machine issues. By using NetLogo, traffic signal plans are implemented with quite a few lines of code to change traffic signal lights according to each plan duration. Although this ad-hoc implementation fits the feature specification, it may not be reusable in other domains. Flow control and state machine agent capabilities, in MDD4ABMS, are domain-independent abstractions. Additionally, some participants mentioned that due to their programming skills, it was easy to follow NetLogo examples because they easily recognized code structures (e.g., conditional statements and function calls). Given that this was their first contact with MDD4ABMS. Additional practice may be necessary to become fluent in its modeling language and tool.

To investigate the effect of feature TF4 on the time it took participants to develop traffic simulations, Figure 5.9 shows the overall time to develop only features TF1—TF3 and TF5. The difference between MDD4ABMS and NetLogo becomes more evident. The time it took to develop these features by using MDD4ABMS (29.72min, on average) is 33.92% less than by using NetLogo (44.98min, on average) when finished simulations are considered. When only the entirely correct simulations are considered, by using MDD4ABMS it took participants 42.98% less time (26.27min, on average) than by using NetLogo (46.03min, on average). A Kruskal-Wallis test revealed significant differences between MDD4ABMS and NetLogo in the case of finished simulations ($\chi^2 = 10.00$, $p < 0.01$) as well as in the case of the entirely correct simulations ($\chi^2 = 9.63$, $p < 0.01$). This result emphasizes the importance of reducing the abstraction gap by means of language constructs at the right abstraction level. Though flow control and state machine

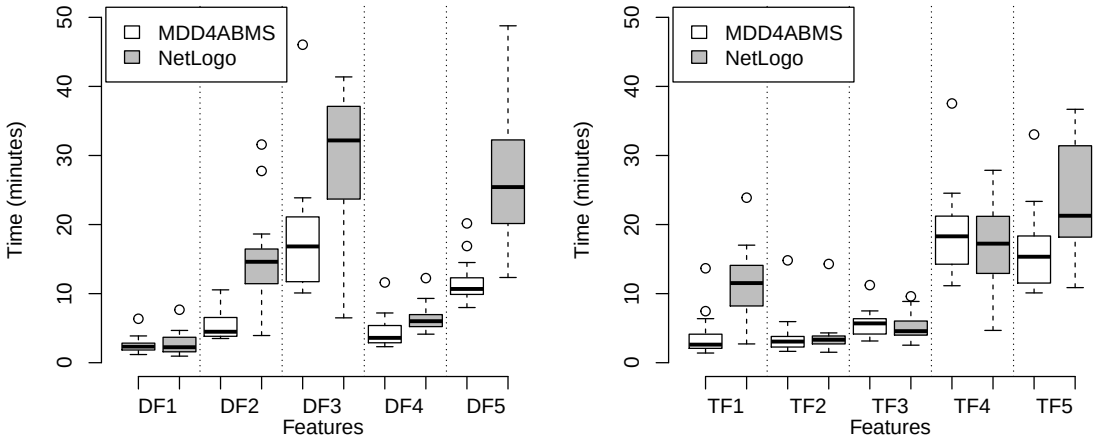
Figure 5.9: Overall Time to Develop Features TF1–TF3 and TF5 in the Traffic Simulation



capabilities are abstract representations for recurrent agent-based simulation aspects, results suggest that the provided notation may not have been enough to decrease the effort to develop traffic signal plans.

Finally, Figure 5.10 presents the time it took participants to develop features considering finished simulations, which include those simulations that are not entirely correct. It can be seen that results resemble the development time of features from entirely correct simulations, previously shown in Figures 5.8(a) and 5.8(b). Once again, the difference between MDD4ABMS and NetLogo is more observable in features that demand sophisticated constructs.

Figure 5.10: Time to Develop Features Considering Finished Simulations



(a) Disease (b) Traffic

Findings: Development Effort (RQ2). Results indicate that MDD4ABMS decreases the development effort in comparison to NetLogo. The effort reduction is more evident in features that require sophisticated code constructs when developed with NetLogo. In these cases, abstractions provided by MDD4ABMS reduced the development time, as the participants were able to focus on *which* elements should be included in the simulation, instead of *how* to implement them.

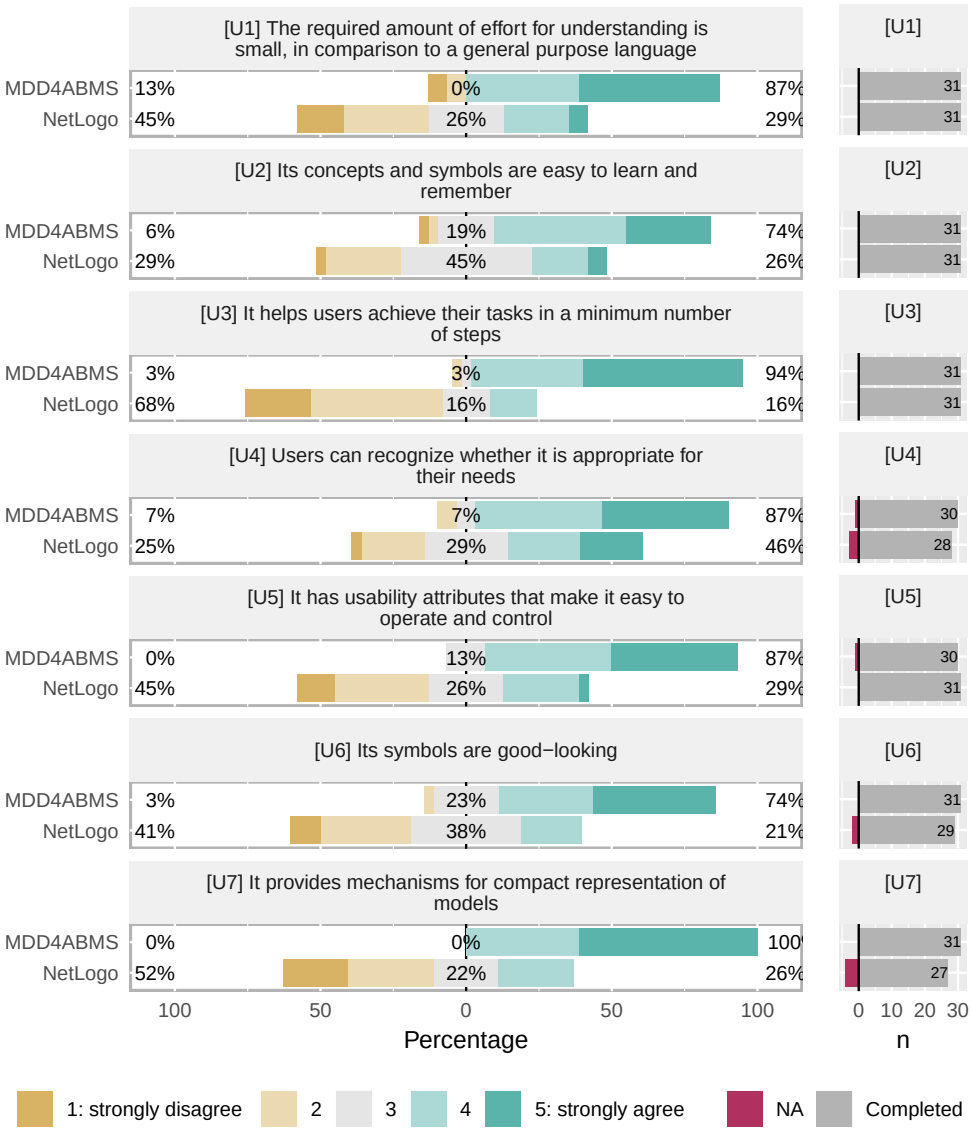
5.3.3.3 Subjective Evaluation (RQ3)

The analysis performed in the previous sections focused on objective measurements collected while participants performed the tasks of our study procedure. Participants were later requested to subjectively evaluate the two target techniques, MDD4ABMS and NetLogo, with respect to qualitative aspects. Obtained answers are summarized in Figures 5.11 and 5.12. In the following plots, the bars indicate the level of agreement with the presented sentences, which are grouped by the aspect being assessed: usability, reliability, productivity, and expressiveness. Statistical tests revealed that MDD4ABMS obtained significantly higher levels of agreement across all measurements. A summary of the statistical tests is presented in Appendix G.

It can be seen that the levels of agreement with sentences associated with usability characteristics are all greater for MDD4ABMS in comparison with NetLogo. More than 80% of participants agreed (above the intermediate level) that the effort for understanding MDD4ABMS is reduced in comparison to a general purpose language (**U1**), it helps users to achieve their tasks in fewer steps (**U3**), its users can recognize whether it is appropriate for their needs (**U4**), it is easy to operate (**U5**) and provides compact representation of models (**U7**)—the latter is the usability characteristic in which MDD4ABMS was best assessed: for all the participants, the level of agreement is above the intermediate level. Results also show that 74% of the participants agreed (above the intermediate level) that MDD4ABMS concepts and symbols are easy to learn and remember (**U2**) and its symbols are good-looking (**U6**). With respect to NetLogo, the levels of agreement on the presence of characteristics **U3** and **U7** are the lowest, meaning that participants found it laborious to develop simulations with NetLogo and that its model representation is less compact.

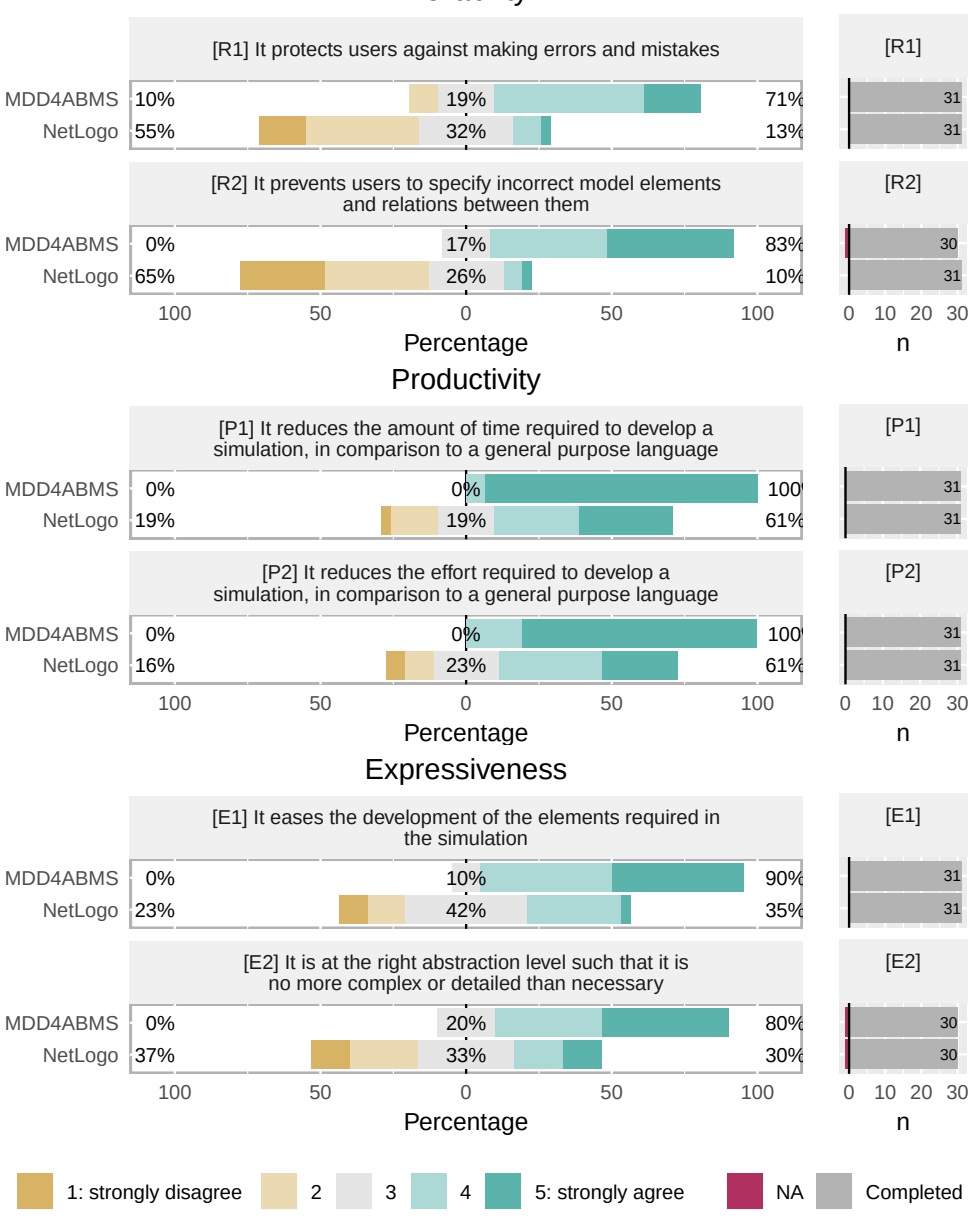
Similar results are observed in the other qualitative aspects, as shown in Figure 5.12. For MDD4ABMS the levels of agreement on reliability characteristics—model checking to protect against errors (**R1**) and correctness (**R2**)—are above the intermediate

Figure 5.11: Subjective Evaluation: Usability



level for more than 70% of participants. In contrast, the levels of agreement for NetLogo show that, for many participants, it does not provide an effective means to avoid error making. Results previously presented regarding the presence of features with syntactical errors (Table 5.10) confirm this user perception, given that no syntactical error was found in simulations developed with MDD4ABMS. The levels of agreement on the presence of productivity characteristics show that both techniques improve the development time (**P1**) and development effort (**P2**) in comparison to developing agent-based simulations using general purpose languages. However, for both of these characteristics, the levels of agreement for MDD4ABMS are greater than for NetLogo, which means that, from the point of view of participants, MDD4ABMS improves time and effort more than NetLogo (i.e. with MDD4ABMS simulations are developed in less time and with less effort), with re-

Figure 5.12: Subjective Evaluation: Reliability, Productivity, and Expressiveness



spect to general purpose programming languages. Finally, with respect to the presence of expressiveness characteristics, 90% of participants agreed (above the intermediate level) that MDD4ABMS eases the development of simulation elements (**E1**) and 77% agreed that MDD4ABMS is at the right abstraction level (**E2**).

Lastly, the main positive aspects reported by participants for MDD4ABMS are that it *eases and promotes agility in the development of simulations*, and that its *abstraction level is higher than NetLogo*. As negative aspects, they reported that *abstractions are limited to the considered application domains*, and that *the lack of tutorials, examples, and documentation may prevent users to learn how to develop simulations with MDD4ABMS by themselves*. With respect to NetLogo, participants reported *it can be*

used to develop simulations in other application domains and *its programming language resembles mainstream programming languages* as its main positive aspects. As negative aspects for NetLogo, they reported that *it demands previous expertise in programming logic* and that *the syntax adopted for some statements differs from mainstream programming languages*.

Findings: Qualitative Aspects (RQ3). Results give evidence that MDD4ABMS meets qualitative aspects related to the user experience, namely usability, reliability, productivity, and expressiveness. MDD4ABMS obtained significantly higher scores with respect to all these measurements than NetLogo, when subjectively evaluated by participants.

5.3.4 Threats to Validity

As in the study previously presented in Section 5.1, this study adopts distinct treatment order for groups (as shown in Table 5.8) to address the threat of participants becoming tired, as well as carry-over effects.

To mitigate the experimenter bias in the procedure adopted for determining at which level features are correct, the set of elements expected for each feature was elaborated prior to the inspection process. That way, all the simulations were inspected considering the same set of expected elements. For NetLogo simulations, the expected elements were specified as logic operations instead of specific programming statements given that developers were free to adopt their preferred coding style. That way, verification is not biased by specific programming statements. For MDD4ABMS simulations, it is expected exactly those metamodel elements that accomplish the simulation model.

Another internal threat is that a few participants dropped out of the study. Participants were free to drop-out from the study at any time. Many participants that have started the mini-course had little or no background on agent-based simulation. Therefore, it was expected that some participants lost their interest in the subject after figuring out exactly what an agent-based simulation is and its purposes. These drop-outs did not compromise the validity of the study because MDD4ABMS— like any other simulation platform—is targeted to people indeed interested in developing agent-based simulations. Participants of the experiment session are the sample that best represents this target audience.

An external threat to validity is that the study considered only two domains (traffic and disease), which may raise issues with respect to the generalization of results. In the presented results, we observed that in both domains there were features in which the development effort reduction was evident due to the high-level abstractions provided by MDD4ABMS. Such abstractions are associated with the creation of agents or the environment initialized with external files (GIS or OSM), and agent capabilities such as reinforcement learning and disease models. This gives evidence that the benefits provided by MDD4ABMS do not depend on the simulation domain, but on the abstraction level provided by the elements of its metamodel and domain-specific modeling language. Despite such evidence, further studies should be carried out in other domains.

Another external threat is that we considered only participants with programming expertise. However, our study goal is not to generalize results for humans in general, but to developers with at least basic knowledge in programming, which matched the characteristics of our sample. Additionally, note that programming expertise favors NetLogo and, therefore, differences could be even larger if the study was performed with participants with less expertise in programming.

5.4 Final Remarks

In this chapter, we presented three empirical studies performed to evaluate MDD4ABMS. Our first study evaluated whether the core metamodel and the ABStractLang language ease the understanding of agent-based simulations. Results show that ABStractLang decreases the time to comprehend agent-based simulations having complex entities and many model parameters in comparison with NetLogo. Our second study used a software engineering metric—effort as a function of the size of the system—to assess the productivity of using MDD4ABMS to develop simulations. Results show that the design and implementation effort required to develop simulation models using MDD4ABMS is lower than models created in the NetLogo or ITSUMO simulation platforms. Finally, our third study evaluated whether the MDD4ABMS approach, when applied to develop agent-based simulation in both the adaptive traffic signal control and spread of disease domains, increases the design quality and decreases the development effort. Results of this study showed that MDD4ABMS indeed promotes expressive modeling, leading developers to create simulations with similar (sometimes better) design quality than NetLogo in less time. Presented studies are novel contributions, given that

previous MDD approaches for ABMS were not evaluated with humans.

Given that the presented domain specific metamodel extensions were built through our proposed domain analysis method, these results also provide evidence that the steps of our domain analysis method are effective and can potentially be used to identify and abstract concepts in other domains.

6 CONCLUSION

The ABMS paradigm has been used to simulate systems in many application areas due to its ability to deals with the complexity that arises from individual behavior and interactions. Developing agent-based simulations is a challenging task. Technical expertise, either in ABMS or in programming logic and simulation platforms may be an issue. Available alternatives for developing simulations range from methodologies, processes, and metamodels, to widely used agent-based simulation platforms. However, these alternatives usually provide support for specifying only elementary aspects of simulations. Sophisticated simulation aspects such as agent capabilities (e.g., learning), disease models, and topologies for the simulated environment, are rarely provided. Hence, developers of agent-based simulation systems must usually implement these aspects from scratch, which can affect productivity and potentially introduce inconsistencies in simulations.

MDD approaches for developing agent-based simulations have been proposed, and this is the context of this thesis. Existing MDD approaches for ABMS cover only elementary aspects of simulations and have issues with respect to expressiveness and code generation. We thus propose MDD4ABMS, an MDD approach that promotes expressive modeling and productivity in agent-based simulation development. The main element of MDD4ABMS is its metamodel. Such a metamodel was built in a bottom-up fashion to abstract aspects recurrently used in simulations. To extend the metamodel with domain-specific aspects, we propose a domain analysis method that considers existing agent-based simulations. In order to promote expressive modeling, we propose ABStractLang, a DSL for modeling simulations. Model-to-code transformations were developed so as to generate code for the NetLogo simulation platform automatically. The ABStractme modeling tool was created to allow specifying models using the ABStractLang language and to generate code from them.

Empirical studies were conducted to evaluate whether MDD4ABMS provide benefits to comprehend simulation models, as well as to develop simulations in two application domains: adaptive traffic signal control and spread of disease. With respect to our research question, obtained results showed evidence that MDD4ABMS reduces the effort in agent-based simulation development with mainstream software developers that have little expertise in ABMS. Regarding our hypothesis, we conclude that MDD is indeed a promising alternative for developers with little expertise in ABMS to ease the development and increase productivity in agent-based simulation development. These studies

also allowed us to identify shortcomings in the MDD4ABMS approach, which will help us to improve our work further. Additional studies can be conducted to refine the results of our studies, tackling identified threats to their validity. Finally, it is worth to mention that this thesis contributed to the popularization of ABMS, given that prior to the mini-courses participants reported little expertise in ABMS-related topics.

6.1 Contributions

As the result of the work presented in this thesis, a number of contributions can be enumerated. Together, these contributions give form to MDD4ABMS, a model-driven approach for developing agent-based simulations.

Metamodel for Agent-based Simulations. The metamodel, described in Chapter 3, is composed of a core and extensions for two application areas: adaptive traffic signal control and spread of disease. The core abstracts elementary aspects of simulations, such as topologies for the simulated environment, in particular grids, Cartesian spaces, and graphs. Additionally, the core also abstracts entities and agents with attributes, as well as basic agent capabilities (mobility and surviving) and data collection. The creation and initialization of both agents and the environment are represented as creational strategies. Strategies for creating those elements according to popular file formats are also provided, such as geospatial files (SANTOS; NUNES; BAZZAN, 2017c). The core metamodel was built considering a set of existing agent-based simulations whose domains are social simulation, land use, epidemic dissemination, and natural disaster application areas. The metamodel was extended through a domain analysis process that considered existing simulations in the adaptive traffic signal control and spread of disease application areas. These extensions capture flow control and decision capabilities (SANTOS; NUNES; BAZZAN, 2017b; SANTOS; NUNES; BAZZAN, 2017a; SANTOS; NUNES; BAZZAN, 2018) and compartmental models that specify disease transmission and dynamics. It should be noted that although this thesis focuses on extensions to these two application areas, our metamodel is not limited in that regard. Aspects abstracted into out metamodel can potentially be adopted to develop simulations in other areas.

Domain Analysis Method. A domain analysis method is proposed to extend the metamodel with aspects recurrently used in specific application areas. The method considers as input existing agent-based simulations and produces as output a list of abstractions that

correspond to the domain model. In the proposed method, the ODD protocol is used to guide the identification of aspects related to agent capabilities and the simulation execution. These aspects are then analyzed, and their underlying essence is abstracted into the metamodel so as to provide ready-to-use building blocks for them. Given that the metamodel extensions proposed in this thesis were constructed following this method, results of the empirical studies also provide evidence that it is effective and can potentially be used to identify and abstract concepts in other domains.

ABStractLang: Domain-specific Modeling Language. The modeling language, described in Chapter 4, provides an expressive means of specifying agent-based simulation models using the abstractions captured by the MDD4ABMS metamodel. ABStractLang adopts a graphical notation and provides building blocks to instantiate elements of the MDD4ABMS metamodel. Given its focus on ABMS, it reduces the abstraction gap and allows expressing *what*, instead of *how*, to consider in models. By providing a modeling language with building blocks for the high-level abstractions captured by the metamodel, as suggested by ABMS researchers, our approach can foster the adoption of the ABMS paradigm.

Model-to-code Transformations. We provided model-to-code transformations to generate code for the NetLogo simulation platform. These transformations, described in Chapter 4, were specified as production rules, which transform elements of our metamodel into NetLogo code statements and blocks. To promote productivity in our MDD4ABMS approach, we automated the execution of such production rules to generate the NetLogo code automatically. Pieces of code for recurrent structures and algorithms are automatically generated, preventing developers from introducing inconsistencies caused by implementing these elements from scratch.

Tool Support: The ABStractme Modeling Tool. We developed ABStractme, a modeling tool that enables developers to create agent-based simulation models in ABStractLang and to generate code from them through the execution of the model-to-code transformations (MOREIRA et al., 2017). ABStractme provides a diagram editor and a palette of components that provides the constructs of the ABStractLang language. [1]

Empirical Evaluations. MDD4ABMS was evaluated by means of three empirical stud-

[1] We thank Deividi Moreira (former computer science undergraduate student), Matheus Cezimbra Barbieri (former scientific initiation student), and Josué Keglevich (scientific initiation student) for their contributions to the development of the modeling tool.

ies, described in Chapter 5 (SANTOS; NUNES; BAZZAN, 2017c; SANTOS; NUNES; BAZZAN, 2018). Our first study was focused on the core metamodel, and considered the graphical notation adopted by ABStractLang for instantiating the abstractions provided by the metamodel. The goal was to evaluate whether ABStractLang eases the comprehension of simulations by humans, and involved 26 participants. Results show that ABStractLang decreases the time to comprehend agent-based simulations having complex entities and many model parameters in comparison do NetLogo. Our second study used a software engineering metric—effort as a function of the size of the system—to assess the productivity of using MDD4ABMS to develop simulations. The effort to create adaptive traffic signal control simulations with MDD4ABMS was compared to the effort to create simulations with NetLogo and ITSUMO. Results show that the design and implementation effort required to develop simulations with MDD4ABMS is lower than with models created in these platforms. Finally, our third study involved 52 participants to evaluate whether the MDD4ABMS approach, when used to develop agent-based simulation in both the adaptive traffic signal control and spread of disease domains, increases the design quality and decreases the development effort. Results of this study showed that MDD4ABMS indeed promotes expressive modeling, leading mainstream software developers to create simulations with similar (sometimes better) design quality than NetLogo in less time. These studies are novel contributions given that previous MDD approaches for ABMS were not evaluated with humans.

6.2 Future Work

The contributions presented in this thesis advance research on model-driven agent-based simulation development. However, our work has limitations, leading to ongoing and future work discussed as follows.

Metamodel Abstractions for MAS and AI Techniques. The metamodel proposed in this thesis captures state machine and reinforcement learning techniques, as well as mobility and surviving agent capabilities, found through the domain analysis processes conducted in the adaptive traffic signal control and spread of disease domains. Plenty of additional MAS and AI techniques are available in the literature (e.g., organizational models, multi-agent reinforcement learning, sentiment analysis, and deep learning). These are currently not abstracted in our metamodel because they were not used in the simulations

considered in the domain analysis. A suggestion for future work is to extend the metamodel to capture such abstractions, allowing developers to use these techniques in an expressive and productive fashion. To use a given model or technique in a particular context, its inputs, as well as any other element required, must be specified in terms of that context. Consider for example the *Q-learning* reinforcement learning technique. The specification of states and actions depends on whether reinforcement learning is adopted for traffic signal controller or vehicle agents. Such a specification depends on the application domain as well: the modeling of states and actions for human agents (e.g., drivers) in the adaptive traffic signal control domain may not be the same for human agents (e.g., patients) in the spread of disease domain. Metamodel profiles can be adopted to handle this. To build such profiles, a cross-domain analysis should be conducted to identify which and how MAS and AI techniques are recurrently used in diverse application domains. Furthermore, such a cross-domain analysis can potentially provide developers with principles for choosing which technique is suitable to produce a particular simulation output. Lack of such principles is pointed out by Wellman (2016) as an issue in the ABMS paradigm.

Metamodel Abstractions for Additional Aspects and Application Domains. In this thesis, two application domains were analyzed to identify aspects recurrently used in simulations and abstract them into the proposed metamodel. Although the provided abstractions have promoted design quality and productivity as experimental evaluations have shown, some aspects of these application domains were left out of the scope of this thesis. For example, vector-borne diseases and intelligent driver models. Agent-based simulations have been used in many other application areas, as mentioned in Chapter 2. With additional domain-specific abstractions and ABStractLang constructs for using them in models, our MDD4ABMS approach would potentially decrease the effort to develop simulations in other application areas as well.

Reuse of Concerns. Reuse of features and behaviors is key to improve productivity in software development. Aspects captured by the MDD4ABMS metamodel introduces the reuse of environment topologies, creational strategies, and agent capabilities recurrently used in simulations. Different simulations within an application domain may demand a group of related elements. For example, a simulation to study the spread of disease over two distinct populations would demand the specification of a concern composed of two agents and a disease model. To provide reuse at the concern level is suggested as future work. External files could be used to provide parameters and to initialize elements of the

reused concerns.

Metamodel and ABStractLang Improvements to Bridge Abstraction Gap Results from the empirical study in which MDD4ABMS was used to develop simulations in the traffic signal control domain—presented in Chapter 5—showed that, although all the simulation aspects were correctly specified using the abstractions provided by the metamodel and with the ABStractLang notation, the abstraction level adopted for some of them (in particular, traffic signal plans specified as state machines) may not have been enough to decrease the effort to develop simulations. In contrast, in the spread of disease domain, a customized state machine was introduced into the metamodel to specify disease models explicitly, and ABStractLang was extended with particular notation for such state machine, bridging the abstraction gap between the simulation model and the application domain. This should be considered when extending MDD4ABMS with aspects from additional domains, reinforcing the demand for using a bottom-up domain analysis method that departs from existing simulations.

Flexible Agent Behavior. MDD approaches for mainstream software are based on assumptions regarding aspects inherent from the application domain. Our model-to-code transformations are based on a particular assumption regarding agents: the behavior of an agent results from the activation of its capabilities in a specific sequence at each timestep. The activation sequence is as follows: decision capabilities, flow control (if exists), mobility, surviving, and then other external capabilities. This specific sequence is derived from the existing simulations considered in the domain analysis. When additional agent capabilities were incorporated into the metamodel, to customize such activation sequence may be necessary. For example, with a *commute* capability, the developer may want to customize what activities the agent does before, during, and after commuting.

Multi-platform Model-to-code Transformations. Our model-to-code transformations produce source code for the NetLogo simulation platform. Despite its popularity—statistics show that the majority of agent-based simulations shared at the CoMSES model library are written in NetLogo (ROLLINS et al., 2014)—other platforms are quite active in the ABMS community as well, such as GAMA and Repast. With model-to-code transformations that produce code for other platforms, our MDD4ABMS approach would also promote *portability* of simulation models. This would benefit developers that prefer running their simulations in platforms other than NetLogo.

Model-to-model Transformations and Model Checking. Support for specifying

models of systems and generating code from them is one of the aspects considered in model-driven approaches. Given its focus on models, Model-driven Development (MDD) opens up possibilities for model-to-model transformations and model checking. As we were interested primarily in expressive modeling and productivity in simulation development, these aspects are not in the scope of this thesis. Model-to-model transformations would allow converting simulation models across simulation paradigms, such as from the Discrete Event System Specification (DEVS) formalism to ABMS, and round-trip engineering could be used to synchronize these models. Model checking would allow validating the agent model to detect inconsistent behaviors before generating its source code (basic checks performed by the ABStractme tool assert consistency of the model structure only). Further formal metamodel specification may be required for that, for example by using temporal logic (DIX; FISHER, 2013).

Tutorials and Model Library. As reported by participants of the user study, lack of tutorials on how to develop simulations may hinder adoption of MDD4ABMS. Ongoing work already addresses this issue.[2] The provision of a model library, with a rich set of examples of agent-based models, as well as wizards to specify basic, predefined, simulation elements according to the application domain, certainly will help to foster adoption of MDD4ABMS.

In summary, the work presented in this thesis advances work on model-driven agent-based simulation development and pave the road towards effective use of MDD in the ABMS paradigm. There is still much to do in order to fully exploit the benefits of MDD for developing simulation in other application areas, as well as to use additional MAS and AI techniques in simulations seamlessly, but our work consists of a significant step towards this.

[2] <http://www.inf.ufrgs.br/prosoft/projects/mdd4abms/tutorials>

REFERENCES

ABDOOS, M.; MOZAYANI, N.; BAZZAN, A. L. C. Traffic light control in non-stationary environments based on multi agent q-learning. In: **2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)**. [S.l.: s.n.], 2011. p. 1580–1585. ISSN 2153-0009.

ABRAHÃO, S.; GÓMEZ, J.; INSFRAN, E. Validating a size measure for effort estimation in model-driven web development. **Information Sciences**, v. 180, n. 20, p. 3932–3954, 2010.

AGRAWAL, M.; CHARI, K. Software effort, quality, and cycle time: A study of CMM level 5 projects. **IEEE Transactions on Software Engineering**, v. 33, n. 3, March 2007.

ALBRECHT, A. J. Measuring application development productivity. In: **Proceedings of the joint SHARE/GUIDE/IBM application development symposium**. [S.l.: s.n.], 1979. v. 10, p. 83–92.

ATKINSON, C.; KÜHNE, T. Model-driven development: A metamodeling foundation. **IEEE Software**, IEEE, v. 20, p. 36–41, 2003.

BADHAM, J.; GILBERT, N. Personal protective behaviour during an epidemic. In: **Social Simulation Conference**. Barcelona: [s.n.], 2014.

BANDINI, S.; MANZONI, S.; VIZZARI, G. Agent based modeling and simulation: An informatics perspective. **Journal of Artificial Societies and Social Simulation**, v. 12, n. 4, p. 4, 2009. ISSN 1460-7425. Available from Internet: <http://jasss.soc.surrey.ac.uk/12/4/4.html>.

BASILI, V.; SELBY, R.; HUTCHENS, D. Experimentation in software engineering. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 12, n. 7, p. 733–743, 1986. ISSN 0098-5589.

BAUER, B.; ODELL, J. UML 2.0 and agents: how to build agent-based systems with the new UML standard. **Engineering Applications of Artificial Intelligence**, v. 18, n. 2, p. 141–157, 2005.

BAZZAN, A. L. C. Opportunities for multiagent systems and multiagent reinforcement learning in traffic control. **Autonomous Agents and Multiagent Systems**, v. 18, n. 3, p. 342–375, June 2009.

BAZZAN, A. L. C.; KLÜGL, F. A review on agent-based technology for traffic and transportation. **The Knowledge Engineering Review**, FirstView, p. 1–29, 4 2013. ISSN 1469-8005.

BELSARE, A. V.; GOMPPER, M. E. A model-based approach for investigation and mitigation of disease spillover risks to wildlife: Dogs, foxes and canine distemper in central india. **Ecological Modelling**, v. 296, p. 102–112, 2015.

BERNON, C. et al. A study of some multi-agent meta-models. In: **International Workshop on Agent-Oriented Software Engineering**. New York, USA: Springer, 2004. (Lecture Notes in Computer Science), p. 62–77.

BETTIN, J. Measuring the potential of domain-specific modeling techniques. In: **Second Domain-Specific Modelling Languages Workshop (OOPSLA)**. Seattle, Washington: [s.n.], 2002. p. 39–44.

BOCCIARELLI, P.; D'AMBROGIO, A. Model-driven method to enable simulation-based analysis of complex systems. In: GIANNI, D.; D'AMBROGIO, A.; TOLK, A. (Ed.). **Modeling and Simulation-Based Systems Engineering Handbook**. [S.l.]: CRC Press, 2014. p. 119–148.

BOEHM, B. et al. Cost models for future software life cycle processes: COCOMO 2.0. **Annals of Software Engineering**, v. 1, n. 1, p. 57–94, 1995.

BURGUILLO-RIAL, J. C. et al. History-based self-organizing traffic lights. **Computing and Informatics**, v. 28, n. 2, p. 157–168, 2012.

CHALLENGER, M.; KARDAS, G.; TEKINERDOGAN, B. A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems. **Software Quality Journal**, p. 1–41, 2015. ISSN 1573-1367.

CHEN, B.; CHENG, H. H. A review of the applications of agent technology in traffic and transportation systems. **IEEE Transactions on Intelligent Transportation Systems**, v. 11, n. 2, p. 485–497, June 2010. ISSN 1524-9050.

CoMSES Net. **OpenABM Computational Model Library**. 2018. <http://www.openabm.org>, Acesso em: Jun/2018. Available from Internet: <http://www.openabm.org>.

CONSEL, C. et al. A generative programming approach to developing dsl compilers. In: GLÜCK, R.; LOWRY, M. (Ed.). **4th International Conference Generative Programming and Component Engineering (GPCE)**. Tallinn, Estonia: Springer Berlin Heidelberg, 2005. p. 29–46. ISBN 978-3-540-31977-1.

COOLS, S.-B.; GERSHENSON, C.; D'HOOGHE, B. Self-organizing traffic lights: A realistic simulation. In: PROKOPENKO, M. (Ed.). **Advances in Applied Self-Organizing Systems**. London: Springer, 2013. p. 45–55.

COUTINHO, L. R. et al. Model-driven integration of organizational models. In: LUCK, M.; GOMEZ-SANZ, J. J. (Ed.). **Proceedings of the 9th International Workshop on Agent-Oriented Software Engineering (AOSE 2008)**. Estoril, Portugal: Springer, 2009. p. 1–15.

CROOKS, A. **GIS and Agent-Based Modeling: Exploring Geographical Information Science (GIS) and Agent-Based Modeling (ABMS)**. 2018. <http://www.gisagents.org>, Acesso em: Jun/2018. Available from Internet: <http://www.gisagents.org>.

CROOKS, A. T.; HAILEGIORGIS, A. B. An agent-based modeling approach applied to the spread of cholera. **Environmental Modelling & Software**, v. 62, p. 164–177, 2014.

CROOKS, A. T.; WISE, S. GIS and agent-based models for humanitarian assistance. **Computers, Environment and Urban Systems**, Elsevier, v. 41, p. 100–111, 2013.

DIX, J.; FISHER, M. Specification and verification of multi-agent systems. In: WEISS, G. (Ed.). **Multi-Agent Systems**. 2. ed. [S.l.]: MIT Press, 2013. chp. 14, p. 641–694.

DUARTE, J. N.; LARA, J. de. ODiM: A model-driven approach to agent-based simulation. In: **Proceedings of the 23rd European Conference on Modelling and Simulation**. [S.l.: s.n.], 2009. p. 158–165.

EDMONDS, B. The use of models - making mabs more informative. In: **International Workshop on Multi-Agent-Based Simulation: Second International Workshop, MABS 2000 Boston, MA, USA, July Revised and Additional Papers**. Boston, USA: Springer, 2001. (Lecture Notes in Computer Science), p. 15–32.

EISINGER, D.; THULKE, H.-H. Spatial pattern formation facilitates eradication of infectious diseases. **Journal of Applied Ecology**, Blackwell Publishing Ltd, v. 45, n. 2, p. 415–423, 2008.

EPSTEIN, J.; AXTELL, R. **Growing Artificial Societies Social Science From The Bottom Up**. [S.l.]: MIT Press, 1996.

EPSTEIN, J. M. Why model? **Journal of Artificial Societies and Social Simulation**, v. 11, n. 4, p. 12, 2008. ISSN 1460-7425. Available from Internet: <http://jasss.soc.surrey.ac.uk/11/4/12.html>.

FENTON, N.; BIEMAN, J. **Software metrics: a rigorous and practical approach**. 3. ed. Boca Raton: CRC press, 2014. 617 p.

FUENTES-FERNÁNDEZ, R. et al. Application of model driven techniques for agent-based simulation. In: **Proceedings of the 8th International Conference on Practical Applications of Agents and Multiagent Systems (PAAMS)**. [S.l.]: Springer, 2010. p. 81–90.

GALÁN, J. M. et al. Errors and artefacts in agent-based modelling. **Journal of Artificial Societies and Social Simulation**, v. 12, n. 1, p. 1, 2009. ISSN 1460-7425.

Galorath Inc. **SEER by Galorath**. 2017. <http://galorath.com/>.

GARRO, A.; PARISI, F.; RUSSO, W. A process based on the model-driven architecture to enable the definition of platform-independent simulation models. In: PINA, N.; KACPRZYK, J.; FILIPE, J. (Ed.). **Simulation and Modeling Methodologies, Technologies and Applications**. [S.l.]: Springer Berlin Heidelberg, 2013, (Advances in Intelligent Systems and Computing, v. 197). p. 113–129. ISBN 978-3-642-34335-3.

GARRO, A.; RUSSO, W. easyABMS: A domain-expert oriented methodology for agent-based modeling and simulation. **Simulation Modelling Practice and Theory**, Elsevier, v. 18, n. 10, p. 1453–1467, 2010.

GERSHENSON, C. Self-organizing traffic lights. **Complex Systems**, v. 16, n. 1, p. 29–53, 2005.

GERSHENSON, C.; ROSENBLUETH, D. A. Adaptive self-organization vs static optimization. **Kybernetes**, Emerald, v. 41, n. 3/4, p. 386–403, Apr 2012. ISSN 0368-492X.

GERSHENSON, C.; ROSENBLUETH, D. A. Self-organizing traffic lights at multiple-street intersections. **Complexity**, Wiley Subscription Services, Inc., A Wiley Company, v. 17, n. 4, p. 23–39, 2012. ISSN 1099-0526.

GHORBANI, A. et al. MAIA: a framework for developing agent-based social simulations. **Journal of Artificial Societies and Social Simulation**, v. 16, n. 2, p. 9, 2013. ISSN 1460-7425.

GHORBANI, A. et al. Model-driven agent-based simulation: Procedural semantics of a MAIA model. **Simulation Modelling Practice and Theory**, v. 49, p. 27–40, dec 2014. ISSN 1569190X.

GÓMEZ-SANZ, J. J.; FERNÁNDEZ, C. R.; ARROYO, J. Model driven development and simulations with the INGENIAS agent framework. **Simulation Modelling Practice and Theory**, v. 18, n. 10, p. 1468 – 1482, 2010. ISSN 1569-190X. Simulation-based Design and Evaluation of Multi-Agent Systems. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S1569190X10001024>.

GÓMEZ-SANZ, J. J.; FUENTES-FERNÁNDEZ, R. Understanding agent-oriented software engineering methodologies. **The Knowledge Engineering Review**, v. 30, p. 375–393, 9 2015. ISSN 1469-8005.

GRIMM, V. et al. A standard protocol for describing individual-based and agent-based models. **Ecological Modelling**, Elsevier, v. 198, n. 1, p. 115–126, 2006.

GRIMM, V. et al. The odd protocol: a review and first update. **Ecological modelling**, Elsevier, v. 221, n. 23, p. 2760–2768, 2010.

HAMILL, L. Agent-based modelling: The next 15 years. **Journal of Artificial Societies and Social Simulation**, v. 13, n. 4, p. 7, 2010. ISSN 1460-7425.

HASSAN, S. et al. Reducing the modeling gap: On the use of metamodels in agent-based simulation. In: **In 6th conference of the european social simulation association (ESSA 2009)**. [S.l.: s.n.], 2009. p. 1–13.

HETHCOTE, H. W. The mathematics of infectious diseases. **SIAM Review**, Society for Industrial and Applied Mathematics, v. 42, n. 4, p. 599–653, 2000.

HOFFERT, J.; SCHMIDT, D. C.; GOKHALE, A. Productivity analysis of the distributed qos modeling language. In: OSIS, J.; ASNINA, E. (Ed.). **Model-Driven Domain Analysis and Software Development: Architectures and Functions**. Hershey, PA, USA: IGI Global, 2011. chp. 8, p. 156–176.

HUTCHINSON, J.; ROUNCEFIELD, M.; WHITTLE, J. Model-driven engineering practices in industry. In: **Proceedings of the 33rd International Conference on Software Engineering**. New York, NY, USA: ACM, 2011. (ICSE '11), p. 633–642. ISBN 978-1-4503-0445-0.

IBA, T.; MATSUZAWA, Y.; AOYAMA, N. From conceptual models to simulation models: Model driven development of agent-based simulations. In: **Proceedings of the 9th Workshop on Economics and Heterogeneous Interacting Agents**. [S.l.: s.n.], 2004. p. 1–12.

ISERN, D.; MORENO, A. A systematic literature review of agents applied in healthcare. **Journal of Medical Systems**, v. 40, n. 2, p. 43, Nov 2015.

JASSS. **How to Submit a Paper to JASSS**. 2017. <http://jasss.soc.surrey.ac.uk/admin/submit.html>. Accessed: 2017-02-22.

JENSEN, R. An improved macrolevel software development resource estimation model. In: **Proceedings of the 5th ISPA Conference**. [S.l.: s.n.], 1983. p. 88–92.

JIN, J.; MA, X. Adaptive group-based signal control using reinforcement learning with eligibility traces. In: **IEEE 18th International Conference on Intelligent Transportation Systems**. [S.l.: s.n.], 2015. p. 2412–2417.

KAHRAMAN, G.; BILGEN, S. A framework for qualitative assessment of domain-specific languages. **Software & Systems Modeling**, v. 14, n. 4, p. 1505–1526, 2015. ISSN 1619-1374.

KARDAS, G. Model-driven development of multiagent systems: a survey and evaluation. **The Knowledge Engineering Review**, Cambridge Univ Press, v. 28, n. 04, p. 479–503, 2013.

KEELING, M. J.; ROHANI, P. **Modeling infectious diseases in humans and animals**. [S.l.]: Princeton University Press, 2008.

KERMACK, W. O.; MCKENDRICK, A. G. Contributions to the mathematical theory of epidemics. ii.—the problem of endemicity. **Proc. R. Soc. Lond. A**, The Royal Society, v. 138, n. 834, p. 55–83, 1932.

KLÜGL, F.; BAZZAN, A. L. C. Agent-based modeling and simulation. **AI Magazine**, v. 33, n. 3, p. 29–40, 2012.

KLÜGL, F.; DAVIDSSON, P. AMASON: Abstract meta-model for agent-based simulation. In: KLUSCH, M.; THIMM, M.; PAPRZYCKI, M. (Ed.). **Multiagent System Technologies**. [S.l.]: Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 8076). p. 101–114. ISBN 978-3-642-40775-8.

KLÜGL, F.; HERRLER, R.; FEHLER, M. SeSAm: Implementation of agent-based simulation using visual programming. In: **Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems**. New York, NY, USA: ACM, 2006. (AAMAS '06), p. 1439–1440. ISBN 1-59593-303-4. Available from Internet: <http://doi.acm.org/10.1145/1160633.1160904>.

KOSAR, T. et al. Comparing general-purpose and domain-specific languages: An empirical study. **ComSIS–Computer Science an Information Systems Journal**, ComSIS Consortium, p. 247–264, 2010.

KRAVARI, K.; BASSILIADES, N. A survey of agent platforms. **Journal of Artificial Societies and Social Simulation**, v. 18, n. 1, p. 11, 2015. ISSN 1460-7425.

KUBERA, Y.; MATHIEU, P.; PICAULT, S. Interaction-oriented agent simulations: From theory to implementation. In: **Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)**. Amsterdam: IOS Press, 2008. p. 383–387.

LI, J.; WILENSKY, U. **NetLogo Sugarscape 3 Wealth Distribution model**. Evanston, IL: [s.n.], 2009. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Available from Internet: <http://ccl.northwestern.edu/netlogo/models/Sugarscape3WealthDistribution>.

LUKE, S. et al. Mason: A multiagent simulation environment. **Simulation**, Sage Publications, v. 81, n. 7, p. 517–527, 2005.

LUND, A. M. Measuring usability with the USE questionnaire. **Usability Interface**, v. 8, n. 2, p. 3–6, 2001.

MACAL, C.; NORTH, M. Introductory tutorial: Agent-based modeling and simulation. In: **Proceedings of the 2014 Winter Simulation Conference**. Piscataway, NJ, USA: IEEE Press, 2014. (WSC '14), p. 6–20.

MACAL, C. M. Everything you need to know about agent-based modelling and simulation. **Journal of Simulation**, v. 10, n. 2, p. 144–156, 2016.

MACAL, C. M.; NORTH, M. J. Tutorial on agent-based modelling and simulation. **Journal of simulation**, Taylor & Francis, v. 4, n. 3, p. 151–162, 2010.

MANNION, P.; DUGGAN, J.; HOWLEY, E. An experimental review of reinforcement learning algorithms for adaptive traffic signal control. In: MCCLUSKEY, L. T. et al. (Ed.). **Autonomic Road Transport Support Systems**. [S.l.]: Springer, 2016. p. 47–66.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. **ACM computing surveys (CSUR)**, ACM, v. 37, n. 4, p. 316–344, 2005.

MINAR, N. et al. **The swarm simulation system: A toolkit for building multi-agent simulations**. Santa Fe, 1996.

MOHAGHEGHI, P.; DEHLEN, V.; NEPLE, T. Definitions and approaches to model quality in model-based software development – a review of literature. **Information and Software Technology**, v. 51, n. 12, p. 1646–1669, 2009. ISSN 0950-5849.

MOODY, D. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 35, n. 6, p. 756–779, nov. 2009. ISSN 0098-5589. Available from Internet: <http://dx.doi.org/10.1109/TSE.2009.67>.

MOREIRA, D. et al. ABStractme: Modularized environment modeling in agent-based simulations. In: DAS, S. et al. (Ed.). **Proceedings of the 16th International Conference on Autonomous Agents and Multiagent Systems**. São Paulo: IFAAMAS, 2017. p. 1802–1804.

MÜLLER, B. et al. Standardised and transparent model descriptions for agent-based models: Current status and prospects. **Environmental Modelling & Software**, Elsevier, v. 55, p. 156–163, 2014.

MURPHY, J. T. **CoMSES Digest: Fall 2015**. 2015. Available from Internet: <https://www.openabm.org/files/CoMSES_Digest_v3_No3.pdf>.

NATHANSON, N. Virus perpetuation in populations: biological variables that determine persistence or eradication. In: PETERS, C. J.; CALISHER, C. H. (Ed.). **Infectious Diseases from Nature: Mechanisms of Viral Emergence and Persistence**. Vienna: Springer, 2005. p. 3–15.

NORTH, M. J.; COLLIER, N. T.; VOS, J. R. Experiences creating three implementations of the Repast agent modeling toolkit. **ACM Transactions on Modeling and Computer Simulation (TOMACS)**, ACM, New York, NY, USA, v. 16, n. 1, p. 1–25, jan. 2006. ISSN 1049-3301.

OLIVEIRA, D. de.; BAZZAN, A. L. C. Multiagent learning on traffic lights control: effects of using shared information. In: BAZZAN, A. L. C.; KLÜGL, F. (Ed.). **Multi-Agent Systems for Traffic and Transportation**. Hershey, PA: IGI Global, 2009. p. 307–321. ISBN 978-160566226-8.

OSIS, J.; ASNINA, E. **Model-Driven Domain Analysis and Software Development: Architectures and Functions**. 1st. ed. Hershey, PA, USA: IGI Global, 2010. ISBN 1616928743, 9781616928742.

OZIK, J. et al. Repast simphony statecharts. **Journal of Artificial Societies and Social Simulation**, v. 18, n. 3, p. 11, 2015. ISSN 1460-7425. Available from Internet: <http://jasss.soc.surrey.ac.uk/18/3/11.html>.

PARKER, M. **Agent Modeling Platform**. 2010. The Eclipse Foundation. Available from Internet: <https://wiki.eclipse.org/Agent_Modeling_Platform>.

PARUNAK, H. V. D.; SAVIT, R.; RIOLO, R. L. Agent-based modeling vs. equation-based modeling: A case study and users' guide. In: SICHMAN, J. S.; CONTE, R.; GILBERT, N. (Ed.). **International Workshop on Multi-Agent Systems and Agent-Based Simulation**. Paris, France: Springer, 1998. (Lecture Notes in Computer Science), p. 10–25.

PAVÓN, J.; GÓMEZ-SANZ, J. Agent oriented software engineering with INGENIAS. In: MAŘÍK, V.; PĚCHOUČEK, M.; MÜLLER, J. (Ed.). **International Central and Eastern European Conference on Multi-Agent Systems**. Prague, Czech Republic: Springer, 2003. (Lecture Notes in Computer Science), p. 394–403.

PUTNAM, L. H.; MYERS, W. **Measures for Excellence: Reliable Software on Time, Within Budget**. 1. ed. [S.l.]: Prentice Hall Professional Technical Reference, 1991. ISBN 0135676940.

RAILSBACK, S. F.; LYTINEN, S. L.; JACKSON, S. K. Agent-based simulation platforms: Review and development recommendations. **SIMULATION**, v. 82, n. 9, p. 609–623, 2006.

RODRÍGUEZ-HERNÁNDEZ, P. S.; BURGUILLO-RIAL, J. C. **HB-SOTL**. 2016. Sourceforge. Available from Internet: <https://sourceforge.net/projects/hb-sotl/>.

ROLLINS, N. D. et al. A computational model library for publishing model documentation and code. **Environmental Modelling & Software**, v. 61, n. 0, p. 59 – 64, 2014. ISSN 1364-8152. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S1364815214001959>.

ROOP, J. **Reinforcement Learning Maze**. NetLogo User Community Models, 2006. Aerospace Systems Design Laboratory (ASDL), Georgia Institute of Technology. Available from Internet: <http://ccl.northwestern.edu/netlogo/models/community/Reinforcement%20Learning%20Maze>.

ROSSETTI, R. J. et al. Using {BDI} agents to improve driver modelling in a commuter scenario. **Transportation Research Part C: Emerging Technologies**, v. 10, n. 5–6, p. 373–98, 2002. ISSN 0968-090X. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0968090X0200027X>.

SANSORES, C.; PAVÓN, J. Agent-based simulation replication: A model driven architecture approach. In: **Proceedings of the 4th Mexican International Conference on Artificial Intelligence (MICAI)**. Monterrey, Mexico: [s.n.], 2005. (Lecture Notes in Computer Sience), p. 244–253.

SANSORES, C.; PAVÓN, J.; GÓMEZ-SANZ, J. Visual modeling for complex agent-based simulation systems. In: **International Workshop on Multi-Agent Systems and Agent-Based Simulation**. [S.l.]: Springer, 2005. p. 174–189.

SANTOS, F.; NUNES, I.; BAZZAN, A. L. C. A case study of the development of an agent-based simulation in the traffic signal control domain using an MDD approach. In: **Proceedings of the 5th International Workshop on Engineering Multi-Agent Systems (EMAS 2017)**. São Paulo: [s.n.], 2017. p. 97–112.

SANTOS, F.; NUNES, I.; BAZZAN, A. L. C. Model-driven engineering in agent-based modeling and simulation: a case study in the traffic signal control domain. In: DAS, S. et al. (Ed.). **Proceedings of the 16th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2017)**. São Paulo: IFAAMAS, 2017. p. 1725–1727.

SANTOS, F.; NUNES, I.; BAZZAN, A. L. C. Supporting the development of agent-based simulations: a DSL for environment modeling. In: **Proceedings of the IEEE Computer Software and Applications Conference (COMPSAC 2017)**. Torino: [s.n.], 2017. 170–179.

SANTOS, F.; NUNES, I.; BAZZAN, A. L. C. Model-driven agent-based simulation development: a modeling language and empirical evaluation in the adaptive traffic signal control domain. **Simulation Modelling Practice and Theory**, v. 83, p. 162–187, April 2018. Available from Internet: <authors.elsevier.com/a/1WiS7,ZhUEIEQf>.

SCHMIDT, D. Model-driven engineering. **Computer-(IEEE Computer Society**, v. 39, n. 2, p. 25–31, feb 2006. ISSN 0018-9162.

SHAO, P.; HU, P. Product diffusion using advance selling strategies: An online social network perspective. **Journal of Artificial Societies and Social Simulation**, v. 20, n. 2, 2017.

SICHMAN, J. S. ao. Operationalizing complex systems. In: FURTADO, B. A.; SAKOWSKI, P. A. M.; TÓVOLLI, M. H. (Ed.). **Modeling Complex Systems for Public Policies**. [S.l.]: IPEA, 2015. p. 85—123.

SILVA, B. C. da. et al. ITSUMO: an intelligent transportation system for urban mobility. In: **Proceedings of the Optimization of Urban Traffic Systems**. Guadalajara, Mexico: Springer-Verlag, 2005. (Lecture Notes in Computer Science), p. 224–235. Available from Internet: <www.inf.ufrgs.br/maslab/pergamus/pubs/4-Silva+2005OUTS.pdf.tar.gz>.

SPRINKLE, J. et al. Guest editors' introduction: What kinds of nails need a domain-specific hammer? **IEEE Software**, v. 26, n. 4, p. 15–18, July 2009. ISSN 0740-7459.

STAHL, T. et al. **Model-driven software development: technology, engineering, management**. [S.l.]: John Wiley & Sons, 2006. 446 p.

STAHL, T. et al. **Model-driven software development: technology, engineering, management**. [S.l.]: John Wiley & Sons, 2006. 446 p.

STREMBECK, M.; ZDUN, U. An approach for the systematic development of domain-specific languages. **Software: Practice and Experience**, John Wiley & Sons, Ltd., v. 39, n. 15, p. 1253–1292, 2009. ISSN 1097-024X.

TAILLANDIER, P. et al. GAMA: A simulation platform that integrates geographical information data, agent-based modeling and multi-scale control. In: DESAI, N.; LIU, A.; WINIKOFF, M. (Ed.). **13th International Conference International Conference Principles and Practice of Multi-Agent Systems (PRIMA 2010**. Kolkata, India: Springer, 2010. p. 242–258.

TOLVANEN, J.-P.; KELLY, S. Model-driven development challenges and solutions: Experiences with domain-specific modelling in industry. In: **Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)**. [S.l.: s.n.], 2016. p. 711–719.

VOELTER, M. et al. **DSL engineering: Designing, implementing and using domain-specific languages**. CreateSpace Independent Publishing Platform, 2013. 558 p. Available from Internet: <http://dslbook.org>.

WATKINS, C. J. C. H.; DAYAN, P. Q-learning. **Machine Learning**, Kluwer Academic Publishers, Hingham, MA, USA, v. 8, n. 3, p. 279–292, 1992. ISSN 0885-6125.

WELLMAN, M. P. Putting the agent in agent-based modeling. **Autonomous Agents and Multi-Agent Systems**, v. 30, n. 6, p. 1175–1189, 2016. ISSN 1573-7454.

WEYNS, D. et al. Agent environments for multi-agent systems–a research roadmap. In: WEYNS, D.; MICHEL, F. (Ed.). **Agent Environments for Multi-Agent Systems IV**. [S.l.]: Springer, 2015, (Lecture Notes in Computer Science, v. 9068). p. 3–21.

WEYNS, D.; OMICINI, A.; ODELL, J. Environment as a first class abstraction in multiagent systems. **Autonomous Agents and Multi-Agent Systems**, v. 14, n. 1, p. 5–30, 2007. ISSN 1573-7454.

WIERING, M. Multi-agent reinforcement learning for traffic light control. In: **Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000)**. [S.l.: s.n.], 2000. p. 1151–1158.

WILENSKY, U. **NetLogo**. 1999. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL. Available from Internet: <http://ccl.northwestern.edu/netlogo/>.

WILENSKY, U. **NetLogo Traffic Grid model**. Evanston, IL: [s.n.], 2003. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Available from Internet: <http://ccl.northwestern.edu/netlogo/models/TrafficGrid>.

WILENSKY, U.; RAND, W. **An introduction to agent-based modeling: modeling natural, social, and engineered complex systems with NetLogo**. [S.l.]: MIT Press, 2015. 504 p.

WOHLIN, C. et al. **Experimentation in Software Engineering**. [S.l.]: Springer, 2012. 236 p.

WOOLDRIDGE, M.; JENNINGS, N. R. Intelligent agents: Theory and practice. **Knowledge Engineering Review**, Cambridge Univ Press, v. 10, n. 2, p. 115–152, 1995.

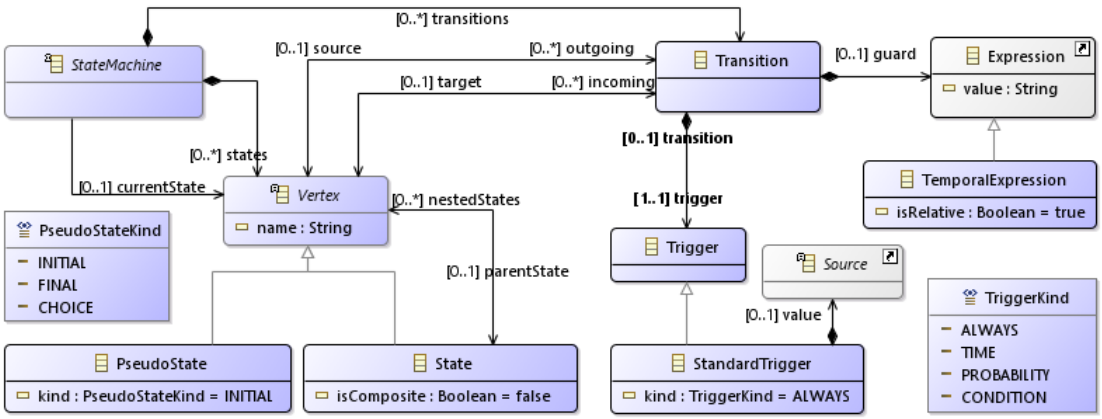
YANG, C.; WILENSKY, U. **NetLogo epiDEM Basic model**. Evanston, IL: [s.n.], 2011. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Available from Internet: <http://ccl.northwestern.edu/netlogo/models/epiDEMBasic>.

YUAN, X.; CROOKS, A. From cyber space opinion leaders and the diffusion of anti-vaccine extremism to physical space disease outbreaks. In: LEE, D. et al. (Ed.). **Proceedings of the 2017 International Conference on Social Computing, Behavioral-Cultural Modeling and Prediction and Behavior Representation in Modeling and Simulation**. Washington: Springer, 2017. (Lecture Notes in Computer Science), p. 114–119.

APPENDIX A — MDD4ABMS METAMODEL: ABSTRACT STATE MACHINE

Figure A.1 shows the abstract state machine elements incorporated into the MDD4ABMS metamodel—highlighted with blue. These are essentially an Ecore representation of a subset of the UML Statemachines metamodel described in the UML specification[1]. In addition to the standard trigger kinds, an additional `PROBABILITY` trigger kind was incorporated to allow specifying probabilistic transitions. Elements from the MDD4ABMS core metamodel described in Section3.2 are highlighted with gray.

Figure A.1: Abstract State Machine Metamodel



[1] <http://www.omg.org/spec/UML/2.5/>

APPENDIX B — MDD4ABMS COMPLETE METAMODEL

Figures B.1 and B.2 shows the complete MDD4ABMS metamodel described in this thesis. The adopted color scheme is as follows.

Basic elements from the core metamodel
Spatial abstraction and spatial units
Relationships
Sources
Creational strategies
Outputs
External agent and agent capability
Surviving agent capability
Mobility agent capability
Flow control agent capability
Decision capabilities
Disease model agent capability
Abstract state machine (Appendix A)

Figure B.1: Complete MDD4ABMS Metamodel: Part 1

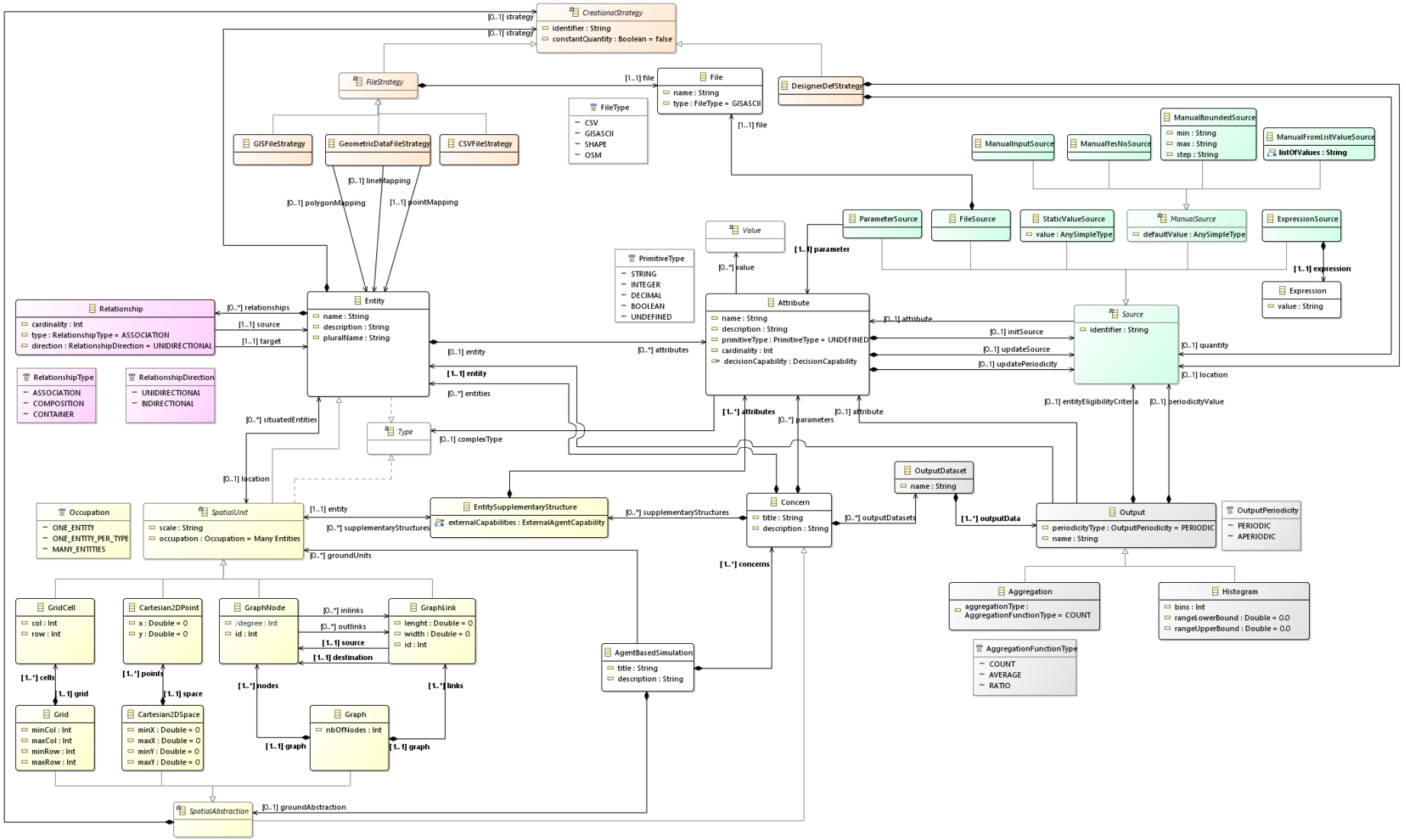
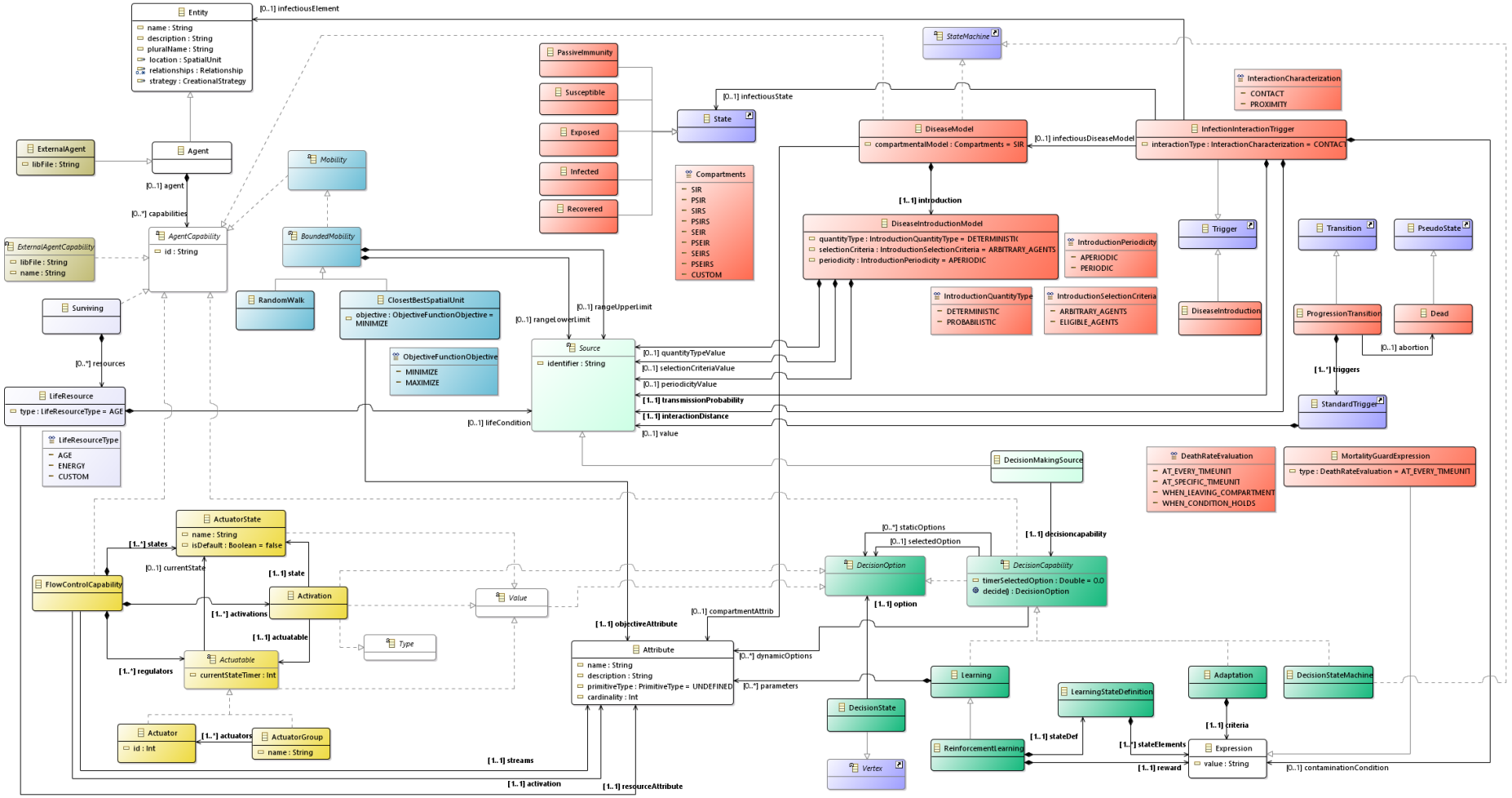


Figure B.2: Complete MDD4ABMS Metamodel: Part 2



APPENDIX C — ABSTRACTLANG CONCRETE SYNTAX FOR MODELING DISEASE ASPECTS

In addition to the elements for modeling the compartmental model and transmissions due to interactions (presented in Section 4.1), the ABStractLang concrete syntax includes elements for modeling disease progression, mortality, and infection introduction.

Figure C.1 shows the concrete syntax for disease progressions. For a given agent disease model `dm`, a progression specification includes the compartment i that is subject to the duration, and the next compartment j. As previously described in Section 3.3.3, this progression is specified as a transition $i \rightarrow j$ in the disease state machine. At least one pair *duration type* and *duration value* (probabilistic, deterministic, conditional) specifies the compartment duration as the transition trigger. A custom duration is specified as more than one pair *duration type* and *duration value*. Specified progressions are shown in a tabular fashion.

Figure C.1: Concrete Syntax of Disease Elements: Progression View

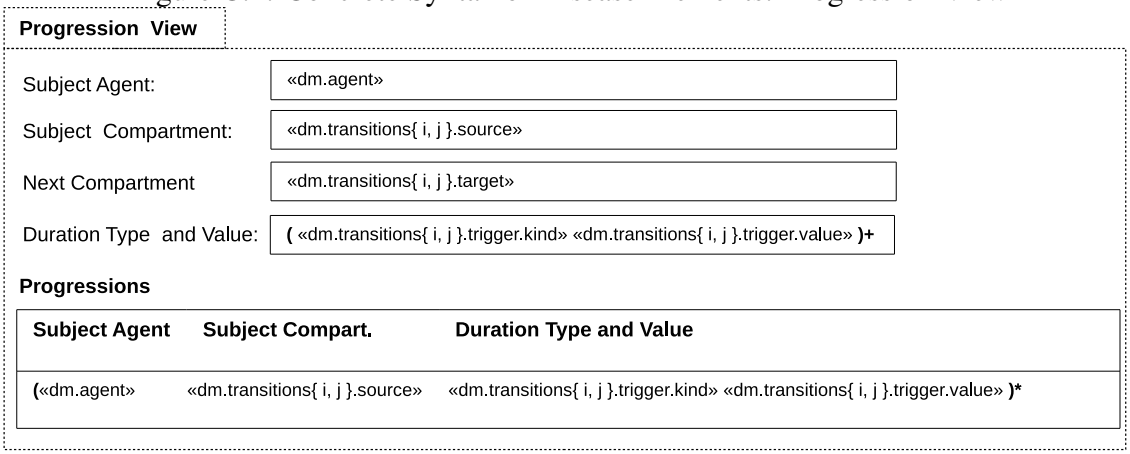


Figure C.2 shows the concrete syntax for specifying deaths caused by the disease. A mortality specification includes the compartment i in which the agent may die due to the disease and a death rate. As previously described in Section 3.3.3, mortality is specified as transitions to the additional, `Dead`, pseudo-state (here represented with the letter `D`). The `mortality type` element model the circumstance in which mortality occurs (*at every timeunit*, *at specific timeunit*, *when condition holds*, or *when leaving compartment*), and it is supplemented with its corresponding value. Specified mortalities are shown in a tabular fashion.

Figure C.2: Concrete Syntax of Disease Elements: Mortality View

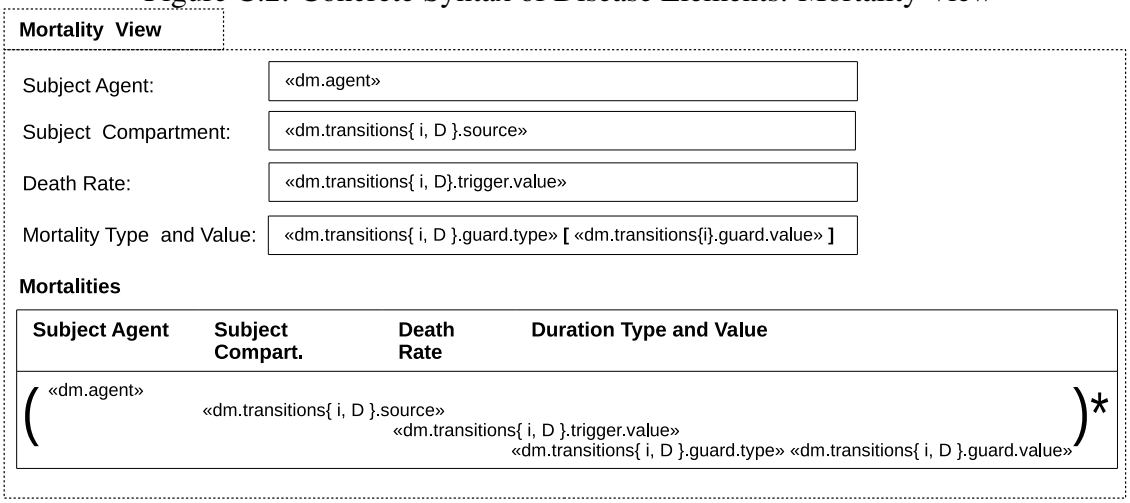
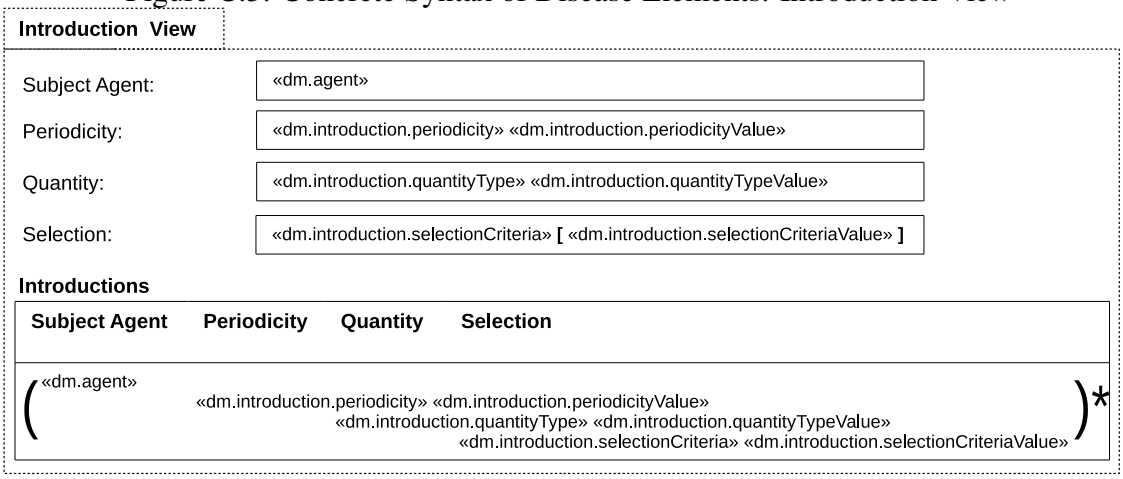


Figure C.3 shows the concrete syntax for modeling infection introduction. An infection introduction specifies periodicity, quantity, and selection criterion according to the predefined types described in Section 3.3.3. Modeled infection introductions are shown in a tabular fashion.

Figure C.3: Concrete Syntax of Disease Elements: Introduction View



APPENDIX D — EXAMPLE OF NETLOGO SOURCE CODE GENERATED BY PRODUCTION RULES

D.1 Source Code for Setting up and Running a Simulation

Listing D.1 shows the NetLogo routines generated to setup and run a simulation. The `setup` routine invokes the generated procedures that create and initialize agents and their capabilities, while the `go` routine, which is activated at every step of the simulation, invokes generated procedures that update agent attributes and activate the agent behavior.

Listing D.1 – NetLogo Generated Routines to Setup and Run the Simulation

```
1  to setup
2      clear-all
3      reset-ticks
4      createSpatialAbstraction
5      initializeGlobals
6      initializePatches
7      createAgents
8      initializeEntityAndAgentBreeds
9      initializeCapabilities
10     VehiclesSetup
11 end
12
13 to go
14     tick
15     updateSpatialUnits
16     updateAgentBreeds
17     ask Traffic_Signal_Control_Agents [
18         Traffic_Signal_Control_Agents-act
19     ]
20     VehiclesGo
21 end
22
23 to Traffic_Signal_Control_Agents-act
24     set activation rl-plan_learning-decide
25     applyFlowControlActivationToActuators activation
26 end
```

D.2 Source Code for the State Machine *gamma plan*

Listing D.2 shows a fragment of the code generated for the state machine `gamma plan`. It is a NetLogo reporter that determines the decision option that must be chosen by the state machine according to the current state and transitions. Any decision capability

of type state machine follows this code structure.

Listing D.2 – Fragment of the Generated Code for the *gamma plan* State Machine

```
to-report sm-beta_plan-decide
    let next_state nobody
    let next_state_init_procedure nobody
     ; finding the transition that is triggered in the current state and its target state
    if sm_beta_plan_0_green = sm_beta_plan_selected_option [
        set next_state eval-transitions sm_beta_plan_0_green_transitions
        set next_state_init_procedure sm_beta_plan_0_green_init_procedure
    ]
    if sm_beta_plan_1_green = sm_beta_plan_selected_option [
        set next_state eval-transitions sm_beta_plan_1_green_transitions
        set next_state_init_procedure sm_beta_plan_1_green_init_procedure
    ]
    ; if any transition held, update the current option and its timer
    ifelse next_state != sm_beta_plan_final_state [
      ifelse next_state = nobody[
          set sm_beta_plan_timer_selected_option sm_beta_plan_timer_selected_option + 1
      ][
          set sm_beta_plan_timer_selected_option 1 ; restart at 1 because the current timestep should be considered
          set sm_beta_plan_selected_option next_state
          run next_state_init_procedure ; initialize the selected option
      ]
    ][
       set sm_beta_plan_selected_option nobody
    ]
    ; reporting the current option
    ifelse is-anonymous-reporter? sm_beta_plan_selected_option [
        report runresult sm_beta_plan_selected_option
    ][
        report sm_beta_plan_selected_option
    ]
end
```

D.3 Source Code for the Learning Capability *plan learning*

Listing D.3 shows a fragment of the code generated for the learning capability `plan learning`. It is a NetLogo reporter that determines the decision option that must be chosen by the learning capability whenever it is activated. As it can be seen, the decision takes into account the computed reward and the current state. Additionally, the `q-table` data structure is updated to reflect what was learned from the last action performed—in other words, the last decision option chosen. Finally, the next decision option is chosen. Because the decision options of this learning capability are state machines, the chosen state machine is activated (with the `runresult` statement) and its decision is reported.

Listing D.3 – Fragment of the Generated Code for the Learning Capability *plan learning*.

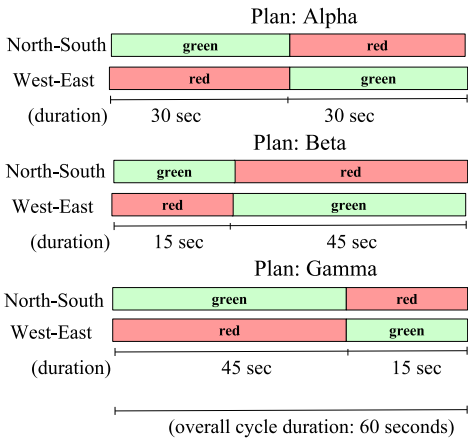
```
to-report rl-plan_learning-decide
    ; a learning capability whose options are state machines only takes a new decision
    ; when no state machine is active, i.e., when the active state machine reaches a terminal state
    if (word rl_plan_learning_selected_option) = (word [ [] -> sm-gamma_plan-decide]) [
        let stateMachineDecision runresult rl_plan_learning_selected_option
        if stateMachineDecision != nobody [
            set rl_plan_learning_timer_selected_option rl_plan_learning_timer_selected_option + 1
            report stateMachineDecision
        ]
    ]
   ; the same verification (ommited here) is performed for each state machine that is decision option of the learning
   ; if no state machine is active, then the learning is activated to report a new decision
   report rl-plan_learning-new-decision
end

to-report rl-plan_learning-new-decision
  ; computing the reward
  let r rl-plan_learning-reward
  ; determining the current state and its position in the states data structure
  let s rl-plan_learning-current-state-from-definition
  set rl_plan_learning_s_idx position s rl_plan_learning_states
  if rl_plan_learning_s_idx = false [
      ; ommited: the state is stored in the q-table if it has not been visited yet
  ]
  ; updating the q-table considering...
  rl-plan_learning-update-qtable
      rl_plan_learning_sprev_idx ; previous state s
      rl_plan_learning_aprev_idx ; previous action a
      rl_plan_learning_s_idx ; current state s'
      r ; reward
  ; action selection, based on the current state s'
  set rl_plan_learning_a_idx rl-plan_learning-select-action rl_plan_learning_s_idx
  ; updating previous state/action
  set rl_plan_learning_sprev_idx rl_plan_learning_s_idx
  set rl_plan_learning_aprev_idx rl_plan_learning_a_idx
  ; finding the concrete action based on the action' index
  let next_decision item rl_plan_learning_a_idx rl_plan_learning_actions
  let next_decision_init_procedure item rl_plan_learning_a_idx rl_plan_learning_action_init_procedures
  ; updating the current action and its timer
  ifelse next_decision != rl_plan_learning_selected_option [
      set rl_plan_learning_timer_selected_option 1    ; restart at 1 since the current timestep should be considered
      set rl_plan_learning_selected_option next_decision
  ][
      set rl_plan_learning_timer_selected_option rl_plan_learning_timer_selected_option + 1
  ]
   ; ommited: if the selected option is a state machine, then it must be restarted
   report runresult rl_plan_learning_selected_option ; reporting the action
end
```

APPENDIX E — DETAILS OF THE SPREAD OF DISEASE SIMULATION DEVELOPED IN THE USER STUDY

<table>
<tr><th>Feature</th><th>Details</th></tr>
<tr><td>Environment</td><td>Grid of 35x35 cells without wrapping.</td></tr>
<tr><td>Native agent</td><td>Movement: at each timestep, it moves to a cell within a radius of two.
Creation and initialization: provided by a GIS file. The file specifies a number of 600 native agents with their initial locations.</td></tr>
<tr><td>Disease in natives</td><td>Disease parameters:
• Compartmental model: SIR.
• Transmission by contact, $\beta = 0.4$.
• Infection duration (τ of the I compartment): 20 timesteps.
• Mortality: death rate $\mu = 0.2$ when the infection duration has elapsed.
• Immunity duration (τ of the R compartment): 33 timesteps.
• Infection introduction: aperiodic, 25 random agents
Simulation outputs:
• Number of native agents,
• Number of susceptible native agents,
• Number of infected native agents,
• Number of recovered native agents,
• All of these observed at every step of the simulation.
Agent coloring: routines provided as a NetLogo library file <code>.nls</code></td></tr>
<tr><td>Immigrant agent</td><td>Movement: same as the Native agent.
Creation and inicialization: a fixed number of 50 agents are created at random locations.</td></tr>
<tr><td>Disease in immigrants</td><td>Disease parameters:
• Compartmental model: SIR.
• Transmission depends on agents' interaction:
<table><tr><th colspan="2">Agent Interaction</th><th></th><th></th></tr><tr><th>Susceptible</th><th>Infected</th><th>β</th><th>Characterization</th></tr><tr><td>Native</td><td>Native</td><td>0.4</td><td>Contact</td></tr><tr><td>Native</td><td>Immigrant</td><td>0.2</td><td>Contact</td></tr><tr><td>Immigrant</td><td>Native</td><td>0.1</td><td>Proximity (distance: 2)</td></tr><tr><td>Immigrant</td><td>Immigrant</td><td>0.3</td><td>Proximity (distance: 1)</td></tr></table>• Infection duration (τ of the I compartment): 15 timesteps.
• Mortality: there is no mortality for the immigrant agents.
• Immunity duration (τ of the R compartment): 40 timesteps.
• Infection introduction: no introduction (all got infected via interactions).
Simulation outputs:
• Number of susceptible immigrant agents,
• Number of infected immigrant agents,
• Number of recovered immigrant agents,
• All of these observed at every step of the simulation.
Agent coloring: routines provided as a NetLogo library file <code>.nls</code></td></tr>
</table>

APPENDIX F — DETAILS OF THE ADAPTIVE TRAFFIC SIGNAL CONTROL SIMULATION DEVELOPED IN THE USER STUDY

Feature	Details
Environment	Graph that represents a traffic network. The traffic network is provided by an OSM file. It has 9 intersection nodes arranged in a 3x3 grid, in addition to 6 input and 6 output nodes through which vehicles enter and exit the network. 24 one-way links interconnect these nodes. Each intersection node has 2 incoming and 2 outgoing links.
Vehicle agent	Vehicles move horizontally or vertically through the network linkss. The vehicle agent is provided as a NetLogo library file. Its attributes and behavior were adapted from the HB-SOTL model by Burguillo-Rial et al. (2012) available at the NetLogo community models library[1]. Additionally, that library implements the creation and insertion of new vehicles in the network according to the following traffic demand: 1200 vehicles/hour are inserted at the north input nodes and move to the south output nodes; and 240 vehicles/hour are inserted at the east input node and move to the east output node.
Traffic signal controller agent	Creation and initialization: 9 agents are created, one at each intersection node. The lanes managed by a particular traffic signal controller are set according the incoming links of the node it is located on.
Traffic signal plans	Three signal plans: 
Reinforcement learning	Learning parameters: • Technique: Q-learning. • Learning states: each state is a pair $(queueLength[1], queueLength[2])$, where $queueLength[i]$ is the number of stopped vehicles on the i^{th} incoming lane. • Learning actions: each signal plan is considered a learning action. The execution of an action corresponds to the activation of its associated signal plan. • Reward function: $0 - (queueLength[1] + queueLength[2])$ • Learning rate: 0.08; Discount factor: 0.25. • Selection policy: Epsilon-greedy. • Epsilon: 0.9; Epsilon-decay: 0.99993 (each step, epsilon decays 0.00007%). Simulation outputs: • Number of stopped vehicles on all lanes, observed at every cycle (60 steps).

APPENDIX G — SUMMARY OF STATISTICAL TESTS: QUALITATIVE ASSESSMENT OF USABILITY ASPECTS

Table G.1: Wilcoxon Signed-rank Tests Comparing MDD4ABMS and NetLogo

Characteristic	V	p-value
U1	305.0	< 0.0009
U2	222.0	< 0.0017
U3	378.0	< 0.0001
U4	179.0	< 0.0050
U5	341.0	< 0.0001
U6	245.0	< 0.0001
U7	325.0	< 0.0001
R1	290.0	< 0.0001
R2	401.5	< 0.0001
P1	231.0	< 0.0001
P2	268.5	< 0.0001
E1	331.5	< 0.0001
E2	349.0	< 0.0008