A Case Study of the Development of an Agent-based Simulation in the Traffic Signal Control Domain using an MDD Approach

Fernando Santos[1][2], Ingrid Nunes[1][3], and Ana L. C. Bazzan[1]

[1] Instituto de Informática, UFRGS, Brazil,
[2] Departamento de Engenharia de Software, UDESC, Brazil
[3] TU Dortmund, Germany
{fsantos,ingridnunes,bazzan}@inf.ufrgs.br

Abstract. Model-driven development (MDD) is an approach for supporting the development of software systems, in which high-level modeling artifacts drive the production of low-level, time and effort-consuming artifacts, such as source code. Previous work on its use showed that it significantly increases development productivity, given that the effort is focused on the business domain instead of technical issues. However, MDD was exploited in the context of agent-based development in a limited way, and previous work has not shown real evidences of the benefits that MDD promotes in this context. In this paper, we explore the use of MDD in agent-based modeling and simulation. We conducted a case study in the traffic signal control domain, in which autonomous agents are in charge of managing traffic light indicators to optimize traffic flow. We propose an MDD approach, composed of a modeling language, and model-to-code transformations for producing runnable simulations. An empirical study provides evidence that our MDD approach reduces the effort to develop agent-based simulations.

Keywords: Agent-based Modeling and Simulation, Model-driven Development, Development Effort, Traffic Signal Control

1 Introduction

Building simulations in which there are multiple interacting agents situated in an environment is a challenging task, which has been widely investigated in the context of agent-based modeling and simulation (ABMS). In ABMS, *modeling* refers to the task of specifying a model that represents a target system (e.g., a traffic system), while *simulation* is the execution of such model over a timeline, running agent behaviors and interactions repeatedly. During the design and implementation of agent-based simulations, different *roles* with distinct expertise must interact and communicate [7], and this is often a barrier to a successful development. Usually, the domain expertise is concentrated on the *thematician* and *modeler* roles, while the technical expertise (on ABMS and its simulation platforms) is concentrated on the *computer scientist* and *programmer* roles. Researchers have already argued about the importance of tools and building blocks that enable domain experts themselves to built agent-based simulations [14].

Many alternatives have been proposed for the design and implementation of agent-based simulations. There are methodologies, languages, and programming platforms [12, 19] that focus exclusively on MAS aspects such as agents, interactions, and the environment. However, those MAS alternatives do not cover simulation aspects, such as the creation and initialization of entities and agents. In turn, agent-based simulation platforms—e.g., NetLogo [31]—are alternatives that consider such simulation aspects. These platforms, however, demand previous expertise in ABMS or in programming, thus hindering domain experts to build agent-based simulations themselves. The demand for technical expertise, such as programming, would significantly reduce with the provision of a solution that enables creating simulations by means of ABMS-related building blocks.

An approach towards this direction is *model-driven development* (MDD), whose goal is to express domain concepts effectively. MDD makes domain concepts (e.g., adaptation) available for modeling by means of a *domain-specific language* (DSL) [22]. Traditional software development approaches, in contrast, often only provide concepts from the solution space (e.g., programming statements and abstract types). Transformation engines and code generators reduce or suppress the development effort when using MDD [25]. There are MDD approaches focused on ABMS, such as metamodels [11, 18] and methodologies to identify the domain concerns that should be included in a modeling language [9, 15]. Nevertheless, they are limited to particular MDD aspects and are abstract in the sense of modeling only high-level ABMS-concepts, leaving much left to be developed in specific applications. Moreover, with such individual specific contributions, there is a lack of evidence of the real benefits that MDD approaches promote, covering the design and implementation of agent-based simulations.

In this paper, we address these issues by further exploring the use of MDD in the context of ABMS and empirically assessing the benefits it provides. We present a case study of the development of an MDD approach in the *adaptive traffic signal control* (ATSC) domain, in which we (i) performed a domain analysis; (ii) designed a metamodel and DSL; and (iii) developed model-to-code transformations. Moreover, we empirically assessed the gains obtained with our approach, in terms of development effort. Our approach is focused on a specific domain, as opposed to existing work that focused on ABMS in general, given that previous work on MDD showed that the more specific the application domain, the higher the chance of success [16]. We describe how the domain concepts were identified through a domain analysis process, which is the foundation of our ABMS metamodel. For the domain analysis, we considered existing simulations, which adopt either adaptation or reinforcement learning techniques, to keep the scope limited due to the aforementioned reasons. Because these techniques are often complex to develop, we provide a DSL that gives building blocks for modeling and automated transformations for code generation. Although we focused on a specific domain, we identified many abstractions, included in our metamodel and DSL, that potentially can be adopted in other domains. We conducted an empirical study that evaluated the reduction of the effort required to develop agent-based simulations. We concluded that our approach reduces the development effort by 60-86%, in terms of produced software assets.

Specifically, our case study provides the following contributions: (i) a domain analysis method, which is applicable to other domains; (ii) a metamodel, DSL, and code generator, which support the development of applications in the investigated domain and potentially in similar domains; and (iii) an evaluation, which concretely demonstrates the benefits of MDD for ABMS.

2 Background on MDD

Software models are widely used in software development. While in traditional model-based software engineering they are used as guidance to source code implementation, in model-driven development (MDD) [25] models are equivalent to source code, because they are used to automatically generate it. Models are thus *first-class citizens*, and the development is driven by modeling artifacts [27].

Making domain abstractions available for modeling is fundamental in MDD. 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 [26]. Such effective MDD approaches are built upon a domain-specific language and its underlying metamodel; and source code generators [27].

A domain-specific language (DSL) is a modeling or programming language designed for a particular domain, trading generality for expressiveness [27]. An MDD approach relies on a DSL to build models [28]. The key element of a DSL—and of an MDD approach as a whole—is its underlying metamodel. Such metamodel defines meta-entities and relationships between them within a domain. The DSL simplifies the metamodel instantiation by providing building blocks for representing model elements and thus a more productive environment. In order to build a DSL, it is crucial to perform a domain analysis [28]. It produces as result a *domain model*, which describes domain concepts that should be provided by the modeling language so as to increase its expressiveness. Such domain model is the basis for creating the DSL metamodel. A DSL is described by its syntax, and semantics. The *abstract syntax* is specified in the DSL metamodel, while the *concrete syntax* specifies the notation used to represent instances of metamodel elements (e.g., textual, or graphical symbols). Finally, semantics specify the meaning of such symbols and their constraints.

Source code generators 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 implemented repeatedly. By putting these pieces of code into code generators, an MDD approach increases productivity in software development. Code generators are build upon production rules that describe what code statements are generated for each metamodel element.

3 MDD for ABMS

As previously stated, in this work we focus specifically on the adaptive traffic signal control (ATSC) domain. This is due to the trade-off to be made in MDD (generality *vs.* expressiveness), and there is evidence that the more specific the application domain, the higher the chance of success [16]. We next detail our case study, first describing the conducted domain analysis, then presenting the resulting metamodel, modeling language, and model-to-code transformations.

3.1 Domain Analysis

Any source of explicit or implicit domain knowledge can be considered in the domain analysis [22]. In this work, we essentially used *existing agent-based simulations* in the ATSC domain. Derived domain concepts were further validated using domain expert knowledge. Consequently, our domain analysis was performed using a proposed *bottom-up* approach, to reduce the bias of individual experts' views while identifying domain concepts. However, we consulted ABMS experts to select ATSC simulations for analysis. As result, we used as source work on self-organizing [5, 10] and reinforcement learning [21, 23, 30].

To guide the domain analysis, we considered existing work in this context, focused on MAS. Hassan et al. [15] proposed a process to guide the identification and formalization of domain concepts. A similar initiative was proposed by Garro and Russo [9]. In spite of providing valuable guidelines for identifying agents, interactions, and environmental entities, these processes do not provide support for identifying the simulation aspects of ABMS, such as temporal extent, initialization, and observation. To overcome this issue, we followed the Overview, Design concepts, and Details (ODD) protocol [13]. This protocol guides 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. Thus, our domain analysis method is composed of the following steps.

Step 1 **Concept Preliminary List.** A list of MAS-related concepts is identified using the existing ATSC simulations (e.g., agents and the environment), following the steps of Hassan et al. [15] and Garro and Russo [9].

Step 2 **ODD-based Refinement.** The ODD protocol is used to refine identified concepts, considering simulation aspects and additional agent capabilities such as learning.

Step 3 **Concept Abstractions.** Identified concepts, already refined based on the ODD protocol, are analyzed in order to find the essence behind them. The analysis considers recurrent characteristics and behaviors. Similar concepts are abstracted as a single, essential concept, or generalized to a parent concept. During the analysis, a table of abstractions is built, containing the domain terminology and concepts, and generalizations.

Step 4 **Domain Modeling.** The table of abstractions is used to build the domain model, which is described in the next section.

After performing the steps described above, which produced intermediate documentation[1], we identified different recurrent concepts in this domain, which are: environment, agents and their capabilities or perceptions, vehicles and demand, adaptation and learning. These are detailed as follows.

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 maps. 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.

The design of a TSC agent involves a set of concepts from the traffic control domain, which comprises our basic domain terminology and is shown in Figure 1. 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 of time 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* corresponds to a complete rotation through all the stages. Consequently, the duration of a cycle corresponds to the sum of its phase intervals. Finally, a *plan* is a set of phases plus the sequence in which they are activated. An offset can be defined for a plan and corresponds to a period of time in which the activation of the first phase is postponed. In order to evaluate the effectiveness of TSC agents, *vehicles* are created in the traffic network during the simulation according to a traffic demand.

Regarding adaptation, each existing simulation considered deals with it in its own particular way. Wiering [30], Oliveira and Bazzan [23], and Mannion et al. [21] described the use of reinforcement learning by TSC agents. Each work adopted a particular state representation and reward function, and the policy learned by TSC agents is related to stage, phase, or plan selection. The work on self-organizing traffic lights [5, 10] introduced a set of adaptive criteria, based on traffic conditions, which drive TSC agent decisions. These introduced adaptation approaches share a common characteristic: TSCs have designer-specified fixed plans, which are used as comparison baselines. Next, we describe how these observations and concepts were abstracted by following steps 3 and 4 of our domain analysis method, to derive the domain model.

3.2 Metamodel and Modeling Language

We next describe the key element of our MDD approach: a metamodel that is in accordance with the concepts identified in our domain model. The DSL concrete syntax and semantics of the modeling language are then detailed, and the semantics is only informally described, due to space restrictions.

[1] http://www.inf.ufrgs.br/prosoft/resources/dsl4abms/2017emas/

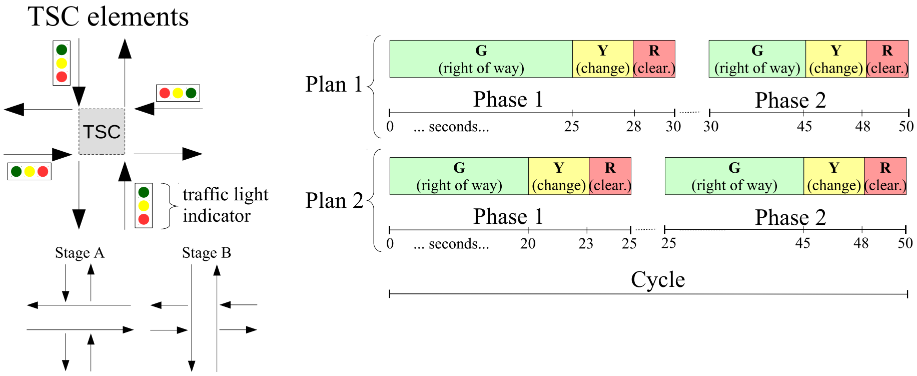


Fig. 1. ATSC Domain Terminology and Concepts.

Metamodel. The metamodel, partially presented in Figure 2, was built following the abstraction step of our domain analysis method. It is constructed upon the EMF Ecore[2] meta-metamodel. The basic elements of the metamodel are entity, agent, and attribute.[3] An *entity* represents objects existing in a simulation (such as lanes and intersections) that has *attributes*. An *agent* is a particular kind of entity that has *agent capabilities*. The idea of agents as entities with attributes are in fact part of almost all agent-based metamodels, such as those presented by Bernon et al. [1]. However, given that we are following a bottom-up approach, existing metamodels are not used as a start point to avoid introduction of bias. Moreover, it can potentially lead to an overly complex metamodel. From the ABMS perspective, such bottom-up specification can provide valuable insights on building an effective MDD approach.

From the abstraction step, we observed that a TSC is, in its essence, an agent that has a *flow control capability* for regulating the flow of a set of streams. Consequently, it has a set of flow regulators for each known stream. These regulators can be seen as *actuators* of the agent. Regulators can be in certain states, such as green or red, open or closed. These states are abstracted to *actuator states*. The actuator state that is automatically activated when no other state is active is the default state. Actuators can be grouped into *actuator groups*, on which all activate the same state simultaneously. Consequently, a group is considered a single actuator because both are considered *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 illustrates how domain concepts are abstracted to these metamodel elements. In the bottom, there are concepts

[2] `http://www.eclipse.org/modeling/emf/`

[3] Element names have the MM prefix in Figure 2 due to the technology used to specify them.

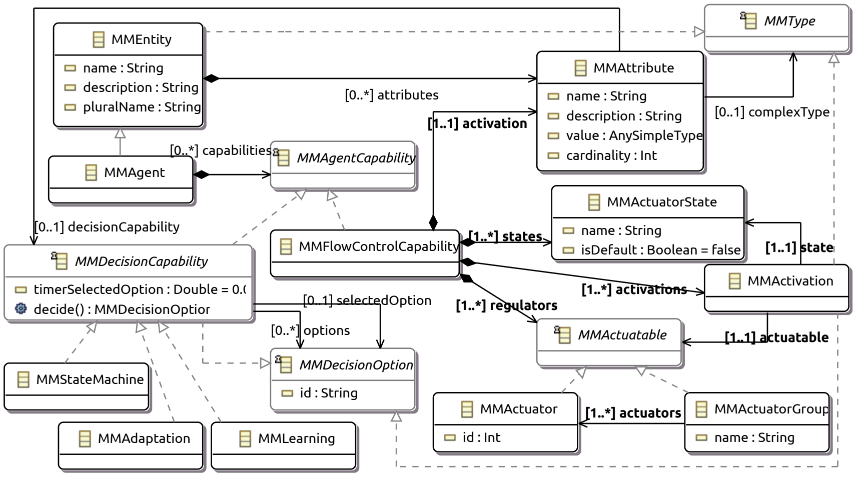


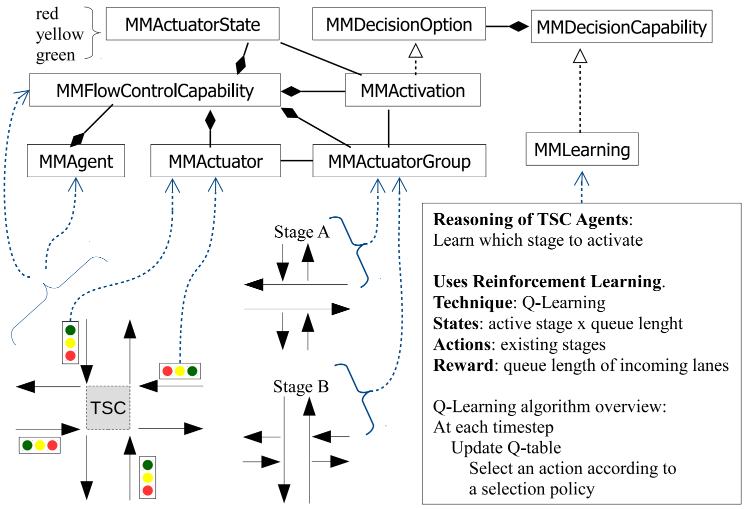
Fig. 2. ABMS Metamodel built based on Existing ATSC Simulations.

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 *flow control capability*. Each traffic signal indicator is an *actuator*, and red/yellow/green states are *actuator states*. Stages are abstracted to *actuator groups* given that the set of actuators that must activate simultaneously is obtained from stage definitions.

The behavior associated with a flow control capability is related to the management of its actuators. At each timestep, an agent must decide which pair (actuator, state) is selected for activation. Such pair is represented as an agent attribute whose type is *activation*. With this attribute, a decision capability must be associated, whose options are all the possible activation pairs. A *decision capability* represents a decision policy that must be chosen from many *decision options*. From the domain analysis, we identified three types of decision capabilities: state machines, adaptation, and learning. It is important to notice that decision capabilities can also be decision options for another decision capability. For example, a learning capability, whose decision options are state machines.

A *state machine* represents a fixed decision policy. It is composed of states, which are pairs of decision option and transitions. To represent state machines, we adopt a subset of the Unified Modeling Language (UML) Statemachines metamodel.[4] An adaptation capability represents an adaptive decision policy. There is an *adaptation criterion* that describes which option should be selected among those available—the one that meets the criteria. Last, a learning capability allows an agent to learn a decision policy. We consider a reinforcement learning capability, with which agents learn through experience. As agents act on the

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



Reasoning of TSC Agents:
Learn which stage to activate

Uses Reinforcement Learning.
Technique: Q-Learning
States: active stage x queue lenght
Actions: existing stages
Reward: queue length of incoming lanes

```
Q-Learning algorithm overview:
At each timestep
   Update Q-table
      Select an action according to
      a selection policy
```

Fig. 3. Example of Concept Abstractions.

environment, they receive a reward signal based on the outcomes of previously states and actions. As illustration, Figure 3 also includes the abstraction of these learning concepts into a learning element. The reasoning of TSC agents is based on reinforcement learning, more specifically on the Q-Learning technique [29].

Language Concrete Syntax. The goal of our language is to ease the modeling of an agent-based simulation by providing building-blocks for its elements. We followed the method of Strembeck and Zdun [28] for building our DSL, whose metamodel was presented in the previous section. We adopted a graphical representation to reduce the effort required to identify model elements and their relationships. Figure 4 shows an overview of the concrete syntax of our language. To illustrate all the language features, the depicted model shows a (partial) combination of existing simulations: stages, phases, and plans as proposed by Oliveira and Bazzan [23]; and the learning technique adopted by Mannion et al. [21].

Entities and agents are represented using a box with at least three sections: the entity or agent name; how it is created; and its attributes. Such representation is inspired by the UML class diagram. The notation used to represent attributes consists of its name and cardinality, followed by the definition of how it is initialized and updated during the simulation. For example, the initialization of the *streams* attribute of the *Traffic Signal Controller* agent is by means of an *expression* that refers to the incoming links of the traffic node; while the updating of the *activation* attribute is related to a decision capability. The entity creation is specified in the second section of their boxes, by means of creational strategies. For the *Traffic Signal Controller* agent, a *designer defined* strategy is selected, in which the designer must specify the quantity of instances and their location. Expressions can be used to specify both of them. We assume

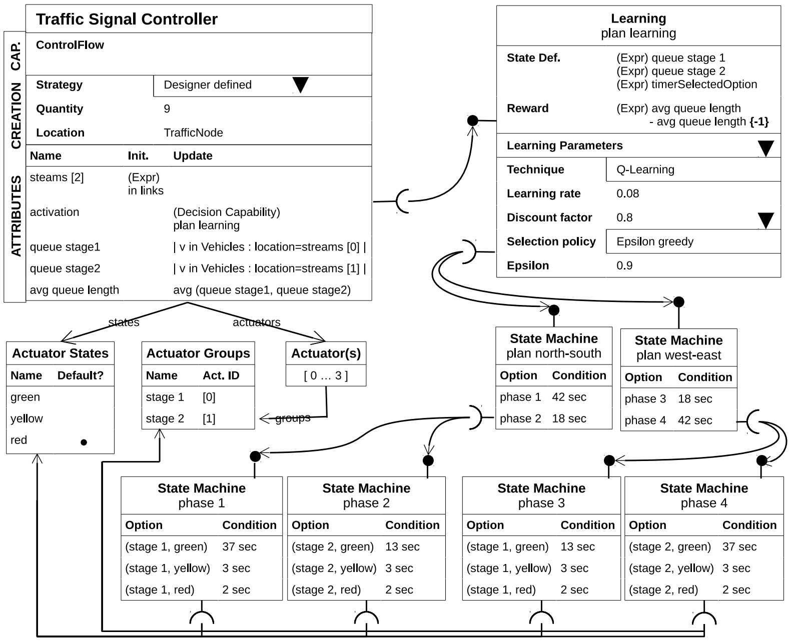


Fig. 4. Example of the Language Concrete Syntax.

that expressions follow a math notation, such as functional or set notations. Metaelements for other creational strategies (i.e., files), and other alternatives for initialization and updating, are included in the complete metamodel, which is not detailed in this paper due to space restrictions.

In addition to the previous sections, an agent box has an additional section to specify its capabilities. Each capability can have a set of predefined attributes that are incorporated into the agent. For example, *Flow Control Capability* incorporates the previously described attributes *streams* and *activation* into the agent. The additional specifications required by an agent capability are represented as elements connected to the agent by an arrow. For the *Flow Control Capability*, its actuator states and actuator groups are represented as tables. Each row in the *Actuator States* table represents a state, and the default state is the one whose *default* column is checked. Similarly, rows in the *Actuator Groups* table represents actuator groups. Each group shown in Figure 4 has only one actuator, because they are used specifically to exemplify group definitions. In such cases, the designer can specify no group and use the identifier of actuators to specify activations, given that one actuator is implicitly created for each stream.

Decision capabilities are represented as boxes whose content describes the elements required by each capability type. The top section of the box presents the corresponding capability type (*State Machine*, *Adaptation*, or *Learning*) and its name for further references. Each capability box includes two connectors, inspired by the UML component diagram. The connector with a semicircle represents that the decision capability requires a set of options, i.e. the decision input. The connector with a filled circle, in turn, represents the decision output, i.e. the selected option. The input provided to a decision capability is represented as a connection between its semicircle connector and any model element that represent decision options, which include actuators, their states and groups, and the output of other decision capabilities. In the model shown in Figure 4, the input of the state machine *phase 1* is the cartesian product of its input connections—actuator states and groups. The output of state machines *phase 1* and *phase 2* are the input of state machine *plan 1*.

For a state machine capability, states are represented as a list of *options* and their transition *conditions*. From all the options available, the designer can select only those that are relevant to the state machine. From the simulations considered in the domain analysis, we assume that the state machine states are activated sequentially. Consequently, each condition specifies the situation that triggers the transition to the state right below it—except for the last state, whose transition is to the first state. A transition condition is specified as an *expression*, which can specify a time interval, in addition to any agent attribute or the current state of the machine and its timer.

Regarding learning capability, its state definition and reward are represented separately from the learning parameters. The state is represented as a list of expressions. For example, the state definition of the *plan learning* capability consists of three expressions that refer to agent attributes. The reward is represented as a single expression. In the *plan learning* example, the reward is defined as the difference between the queue length in the current and previous timesteps. The ability to consider attribute values from previous timesteps in expressions is an additional abstraction incorporated in our metamodel. The adopted notation is to specify the absolute or relative timestep of the attribute enclosed in braces. The learning parameters section presents the selected reinforcement learning technique and their specific parameters. The value of these parameters can be specified statically, or they can refer to expressions or model parameters.

Finally, the representation of an adaptation capability is similar to those of learning and state machines. It introduces only a section for specifying the adaptation criteria as an expression. The model of Figure 4 does not include an adaptation capability; however, its representation is shown in the complete concrete syntax description available elsewhere.[1]

3.3 Model-to-code Transformations

Our metamodel and DSL provide the support needed to model agent-based simulations. However, support for code generation is fundamental to reduce the effort to develop them. In order to show that it is possible to generate runnable simulations from our metamodel, and thus exploit the benefits of an MDD approach, we specified model-to-code transformations to generate code for NetLogo [31], a popular agent-based simulation platform.

Model-to-code transformations are performed through the use of production rules, which transform instantiated concepts of our metamodel to NetLogo code

Table 1. Production Rules (Partial View).

Production Rule	**Transformation to NetLogo code**
agent type	For each *MMAgent* → `breed`
agent attributes	For each *MMAttribute* → `breed-own` For each capability required data structure (e.g., Q-table) → `breed-own`
agent capabilities	For each *MMAgentCapability* → corresponding capability rule
rl capability	For each *MMReinforcementLearning* → **qlearning init.** **qlearning init.** **qlearning reward def.** **qlearnig updt. qtable** **qlearning decision**
qlearning reward def.	For the *MMExpression* reward → `reporter`, which evaluates the reward expression and returns its value
qlearning init.	For each element of *MMLearningStateDefinition* → `set` statements for setting up states For each *MMDecisionOption* → `set` statements for setting up actions
qlearnig updt. qtable	executes reward `reporter` to compute its value `set` statements for updating the `Q-table` considering 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 decision	`reporter`, which selects and reports an action according to a selection policy. For a ϵ-greedy policy, it would be $\text{argmax}_a Q(s, a)$ with probability $1-\epsilon$, and a random action with probability ϵ.

statements and blocks. Due to space restrictions, we are not able to present all production rules needed for our approach. However, Table 1 illustrates some of the rules for transformation of agents, their attributes and capabilities. Rules related to the reinforcement learning capability, more specifically for the Q-Learning [29] technique, are also shown. The production rule column indicates the rule name which, when applied, is transformed into the content presented in the transformation column. 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; `reporter` is used to declare a procedure; and `set` is the assignment statement.

Production rules were implemented as templates using the Xpand[5] template language. Each template describes source code that is generated for its corresponding metamodel element. Generated simulations were verified to assert that it produces the expected behavior.

4 Evaluation

We conducted an empirical study to evaluate the *effort* to develop an agent-based simulation using our approach. As discussed, existing MDD alternatives cover only conceptual ABMS-modeling and, consequently, models must be implemented using existing agent-based simulation platforms or general purpose programming languages to have a runnable simulation. We thus compared our approach to alternatives with which a runnable simulation can be developed.

To select the simulations for our study, we adopted the following criteria: source code availability; coverage of the decision capabilities of our metamodel; and use of NetLogo, if possible, given that our MDD approach generates code

[5] `http://www.eclipse.org/modeling/m2t/?project=xpand`

for this platform. As result, one simulation was selected for each decision capability: (i) fixed traffic signal plans, covering *state machine*; (ii) self-organizing traffic lights [10], covering the aspect of *adaptation*; and (iii) minimization of the number of stopped vehicles [23], covering *learning*. The first two simulations are available for NetLogo[6,7], while the last for the ITSUMO simulation platform[8].

To measure the effort to develop the existing source code, we used the objective metrics proposed in a framework [4] for evaluating languages for multiagent systems. In this framework, the number of manual or automatically produced software artifacts is an objective metric of the development effort. Manual or generated lines of code (LoCs) are examples of these software artifacts [4]. This can be used to measure development effort, as evidenced by many software cost estimation models, such as COCOMO 2.0 [3], which uses size metrics (e.g. LoC) to estimate required person-months and calendar months. Our study considered only code dedicated to TSCs and their behaviors. Comments and code block delimiters were ignored.

To evaluate the effort to develop an agent-based simulation using our MDD approach, we modeled these existing simulations using our DSL and we generated its corresponding source code using our transformation engine. Given that our DSL uses a graphical representation, we used the *atomic model element* (AME) measure unit [2] for a fair comparison. An AME is a visual modeling element that is equivalent to a LoC. To classify a modeling element of our DSL as an AME, we adapted a generous estimation rule [2], based on the elements of UML diagrams. 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 connectors of agent capabilities.

Obtained results are shown in Table 2, separated by decision capability. Columns indicate: (i) **AMEs**: the number of specified AMEs; (ii) **MLoCs**: the number of manually written lines of code; (iii) **Effort**: the sum of AMEs and MLoCs, which is the effort to develop the corresponding simulation; and (iv) **GLoCs**: the number of lines of code generated from AMEs.

As can be seen, our approach reduces the effort for all the three simulations. The development effort was reduced by **60.00%** (with 116 GLoCs), **81.93%** (103 GLoCs), and **85.60%** (297 GLoCs), respectively. For all simulations, TSC-related code was fully automatically generated. From the complete source code required to run a simulation, approximately 50% is automatically generated. Vehicle and visualization-related code, which are out of the scope of our approach, corresponds to the remaining 50%.

It is also possible to observe that the total amount of lines of code (MLoCs + GLoCs) produced by our MDD approach for TCSs and their behavior is greater than the total amount of lines of code in the existing NetLogo implementation, for all simulations. Given that our metamodel considers domain independent ab-

[6] `http://ccl.northwestern.edu/netlogo/models/TrafficGrid`

[7] `http://turing.iimas.unam.mx/~cgg/sos/SOTL/SOTL.html`

[8] `http://www.inf.ufrgs.br/maslab/traffic/itsumo/`

Table 2. Effort Comparison.

Decision Capability	**Simulation**	**AMEs**	**MLoCs**	**Effort***	**GLoCs**
State Machine	NetLogo	1	34	35	1
(Fixed Plan)	Our MDD approach	14	0	14	116
Adaptation	NetLogo	1	82	83	1
(Self-organizing Traffic Lights)	Our MDD approach	15	0	15	103
Learning	ITSUMO	0	257	257	0
(Reinforcement Learning	Our MDD approach	37	0	37	297

*Effort = AMEs + MLoCs

stractions (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, GLoCs are not considered in the development effort metric, because no human effort is required to produce them.

Even if an MDD approach provided code generation for all the elements of a simulation model, it would provide little benefit if the effort to specify such model using the provided DSL is similar to or higher than implementing it from scratch. Based on the presented results, this is not the case of our MDD approach. Given that an AME is equivalent to a MLoC, the sum of AMEs and MLoCs produced when using our MDD approach is always lower than when using NetLogo. Although the number of AMEs produced using our MDD approach is the highest, it is lower than the number of MLoCs produced using NetLogo, leading to a reduction in the combined effort. Furthermore, the rule adopted for counting AMEs led to a fine-grained and platform independent evaluation. Therefore, a few elements of our DSL were counted as AMEs, but the effort to specify them 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 a stronger confidence of that our approach *reduces the development effort* of agent-based simulations in our domain.

5 Related Work

In this section, we discuss work related to our MDD approach. Kardas [17] reviewed a selection of model-driven approaches for MAS. Although there are MDD alternatives that provide modeling languages or source code generation, the author argues that their efficiency and practicability are under debate since the amount and quality of the automatically generated MAS components appear to be insufficient. In most situations, such code is generated only at the template level, and a significant amount of code needs to be completed manually. Additionally, most of the MDD approaches reviewed do not include empirical evaluation [17]. There is a lack of evaluations that go beyond demonstrating the use or feasibility of these approaches and thus lacking real evidences of their concrete benefits. Furthermore, it is important to notice that these alternatives are focused on multiagent models, and thus simulation aspects are uncovered.

Existing MDD approaches for ABMS focus on particular aspects of MDD. The AMASON [18] metamodel covers basic structures and dynamics of agent-based simulations. The MAIA [11] metamodel captures social concepts such as norms and roles. Ribino *et al.* [24] propose a conceptual metamodel to be used as a guideline and concept repository for designing simulations. Overall, these metamodels support only abstract ABMS-concepts, leaving much left to be developed in specific applications. In the IODA methodology [20], behaviors are encoded in a interaction matrix, which can be seen as a DSL for specifying the simulation dynamics. However, there are no available coarse-grained simulation building blocks and thus complex interactions must be specified from scratch. Our MDD approach, in turn, supports concrete concepts and their recurrent characteristics and capabilities (e.g., control flow and decision capabilities) identified using a domain analysis that considered existing simulations.

Transformations for code generation were considered in MDA4ABMS [8]. This work provides a light, task-based metamodel, but no DSL is provided. Instead, the use of UML activity diagrams is proposed for modeling, complemented by guidelines for transforming these diagrams into code artifacts. However, human intervention is required to drive the transformations (e.g., to specify which tasks should be executed at a simulation step). Our MDD approach provides automated code generation that produces runnable simulations.

Finally, MDD alternatives in the context of agent-based traffic simulations were already considered [6]. Their metamodel is based on general, agent-related concepts such as tasks, goals, and facts. To specify a sophisticated behavior (i.e., a learning capability), several elements must be instantiated for modeling it. In our MDD approach with its DSL, such sophisticated structures were reduced to their essence and incorporated in the metamodel as ready-to-use building blocks, reducing the burden of simulation development.

6 Conclusion

Developing agent-based simulations is a challenging task, because of a heterogeneous group of involved roles. Model-driven engineering allows focusing on domain concerns while hiding implementation details. Previous work adopted MDD in the context of agent-based modeling and simulation in a limited way, and lack real evidences of its promoted benefits.

In this paper, we explored the use of this approach in the ABMS context by means of a case study in the adaptive traffic signal control domain. Narrowing the domain is important because in MDD there is a trade-off between generality and expressiveness. As result, we provided a metamodel, domain-specific language and code generation for agent-based simulations in this investigated domain. The work is founded on a domain analysis performed in a disciplined way, using existing agent-based simulations. Steps of our domain analysis allowed us to identify the concepts added to our metamodel. These concepts were identified in our case study and, therefore, we have evidence that they are suitable to our particular investigated domain. Nevertheless, concepts, such as the flow regulator agent, are present in other similar domains, and can be potentially reused. Examples of such domains are the distribution of provisions to relief centers in a disaster simulation, or the regulation of the throughput of links in a data network. Further studies must be performed to validate this.

Our evaluation showed that our approach reduces 60-86% of the development effort. These obtained results provide evidence that our approach gives helpful support for developing agent-based simulations in our domain and that the conducted domain analysis method is effective.

Our long term goal is to use MDD to allow people with little or no ABMS expertise to build agent-based simulations. Further work must be conducted towards this goal. First, experiments with humans must be conducted in order to evaluate subjective aspects, such as usability and comprehensibility. Moreover, specifically in our domain, other simulation techniques and aspects, left out of the scope such as alternative learning models and communication, should be incorporated to our metamodel.

Acknowledgments. This work is supported by funds from CNPq ref. 442582 and 305062, CAPES 7619-15-4, and Alexander von Humboldt BRA 1184533 HFSTCAPES-P.

References

1. C. Bernon, M. Cossentino, M.-P. Gleizes, P. Turci, and F. Zambonelli. A study of some multi-agent meta-models. In *AOSE*, pages 62–77, 2004.
2. J. Bettin. Measuring the potential of domain-specific modeling techniques. In *OOPSLA*, pages 39–44, Seattle, Washington, 2002.
3. B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering*, 1(1):57–94, 1995.
4. M. Challenger, G. Kardas, and B. Tekinerdogan. A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems. *Software Quality Journal*, pages 1–41, 2015.
5. S.-B. Cools, C. Gershenson, and B. D'Hooghe. Self-organizing traffic lights: A realistic simulation. In *Advances in Applied Self-Organizing Systems*, pages 45–55. Springer, London, 2013.
6. A. Fernndez-Isabel and R. Fuentes-Fernndez. Analysis of intelligent transportation systems using model-driven simulations. *Sensors*, 15(6):14116–14141, 2015.
7. J. M. Galán, L. R. Izquierdo, S. S. Izquierdo, J. I. Santos, R. del Olmo, A. López-Paredes, and B. Edmonds. Errors and artefacts in agent-based modelling. *JASSS*, 12(1):1, 2009.
8. A. Garro, F. Parisi, and W. Russo. A process based on the model-driven architecture to enable the definition of platform-independent simulation models. In *SIMULTECH*, pages 113–129. 2013.
9. A. Garro and W. Russo. easyABMS: A domain-expert oriented methodology for agent-based modeling and simulation. *Simulation Modelling Practice and Theory*, 18(10):1453–1467, 2010.
10. C. Gershenson. Self-organizing traffic lights. *Complex Systems*, 16(1):29–53, 2005.

11. A. Ghorbani, P. Bots, V. Dignum, and G. Dijkema. Maia: a framework for developing agent-based social simulations. *Journal of Artificial Societies and Social Simulation*, 16(2):9, 2013.
12. J. J. Gómez-Sanz and R. Fuentes-Fernández. Understanding agent-oriented software engineering methodologies. *The Knowledge Engineering Review*, 30:375–393, 9 2015.
13. V. Grimm, U. Berger, D. L. DeAngelis, J. G. Polhill, J. Giske, and S. F. Railsback. The odd protocol: a review and first update. *Ecological modelling*, 221(23):2760–2768, 2010.
14. L. Hamill. Agent-based modelling: The next 15 years. *Journal of Artificial Societies and Social Simulation*, 13(4):7, 2010.
15. S. Hassan, R. Fuentes-Fernandez, J. M. Galan, A. Lopez-Paredes, and J. Pavon. Reducing the modeling gap: On the use of metamodels in agent-based simulation. In *ESSA*, pages 1–13, 2009.
16. J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In *ICSI*, pages 633–642, 2011.
17. G. Kardas. Model-driven development of multiagent systems: a survey and evaluation. *The Knowledge Engineering Review*, 28(04):479–503, 2013.
18. F. Klügl and P. Davidsson. Amason: Abstract meta-model for agent-based simulation. In *MATES*, pages 101–114. 2013.
19. K. Kravari and N. Bassiliades. A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 18(1):11, 2015.
20. Y. Kubera, P. Mathieu, and S. Picault. Interaction-oriented agent simulations: From theory to implementation. In *ECAI*, pages 383–387, 2008.
21. P. Mannion, J. Duggan, and E. Howley. An experimental review of reinforcement learning algorithms for adaptive traffic signal control. In *Autonomic Road Transport Support Systems*, pages 47–66. Springer, 2016.
22. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.
23. D. de. Oliveira and A. L. C. Bazzan. Multiagent learning on traffic lights control: effects of using shared information. In *Multi-Agent Systems for Traffic and Transportation*, pages 307–321. IGI Global, 2009.
24. P. Ribino, V. Seidita, C. Lodato, S. Lopes, and M. Cossentino. Common and domain-specific metamodel elements for problem description in simulation problems. In *FedCSIS*, pages 1467–1476, 2014.
25. D. Schmidt. Model-driven engineering. *Computer-(IEEE Computer Society*, 39(2):25–31, feb 2006.
26. J. Sprinkle, M. Mernik, J. P. Tolvanen, and D. Spinellis. Guest editors' introduction: What kinds of nails need a domain-specific hammer? *IEEE Software*, 26(4):15–18, July 2009.
27. T. Stahl, M. Völter, J. Bettin, A. Haase, and S. Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2006.
28. M. Strembeck and U. Zdun. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15):1253–1292, 2009.
29. C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.
30. M. Wiering. Multi-agent reinforcement learning for traffic light control. In *ICML*, pages 1151–1158, 2000.
31. U. Wilensky. NetLogo, 1999. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.