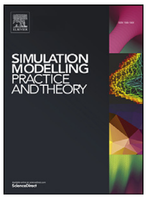
Simulation Modelling Practice and Theory 83 (2018) 162–187

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat



Model-driven agent-based simulation development: A modeling language and empirical evaluation in the adaptive traffic signal control domain

Fernando Santos[a,b,*], Ingrid Nunes[a,c], Ana L.C. Bazzan[a]

[a] *Universidade Federal do Rio Grande do Sul (UFRGS), Instituto de Informática, Avenida Bento Gonçalves, 9500, Porto Alegre, Rio Grande do Sul, 91501-970, Brazil*
[b] *Universidade do Estado de Santa Catarina (UDESC), Ibirama, Brazil*
[c] *TU Dortmund, Dortmund, Germany*

ARTICLE INFO

Article history:
Available online 8 December 2017

Keywords:
Agent-based modeling and simulation
Model-driven development
Development effort
Traffic signal control

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 time and effort-consuming low-level artifacts, such as the source code. Previous studies of the MDD effectiveness showed that it significantly increases development productivity, because the development effort is focused on the business domain rather than technical issues. However, MDD was exploited in the context of agent-based development in a limited way, and most of the existing proposals demonstrated the effectiveness of using MDD in this context by argumentation or examples, lacking disciplined empirical analyses. In this paper, we explore the use of MDD for agent-based modeling and simulation in the adaptive traffic signal control (ATSC) 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 automatically. In order to analyze the productivity gains of our MDD approach, we compared the amount of design and implementation artifacts produced using our approach and traditional simulation platforms. Results indicate that our approach reduces the workload to develop agent-based simulations in the ATSC domain.

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.

* Corresponding author.
E-mail addresses: fernando.santos@udesc.br (F. Santos), ingridnunes@inf.ufrgs.br (I. Nunes), bazzan@inf.ufrgs.br (A.L.C. Bazzan).

https://doi.org/10.1016/j.simpat.2017.11.006
1569-190X/© 2017 Elsevier B.V. All rights reserved.

ABMS has been used to model simulations in many application areas, such as traffic, ecology, economics, and epidemiology [58]. According to Macal and North [58], 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. Additionally, in agent-based simulations, agents can be endowed with learning or evolutionary capabilities to adapt to changes in themselves or the environment. Artificial intelligence techniques that provide such capabilities are well established and can be incorporated in agents leading to more realistic simulations [54].

The development of agent-based simulations involves different roles that must interact and communicate, each role having distinct expertise [30]. 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 (in 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 increase the abstraction level and therefore reduce the required technical expertise [44,54,65,89]. Such approaches would potentially ease the development of agent-based simulations.

Many alternatives have been proposed for supporting the development of agent-based simulations. Agent-based simulation platforms are the most promising, because they consider multiagent system (MAS) aspects such as agents, interactions, and the environment, in addition to simulation aspects such as the creation and initialization of entities and agents. These platforms, however, demand previous expertise in ABMS or programming. This demand for technical expertise would be significantly reduced by 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) [6,78], a software development approach 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) [60]. 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 [78].

MDD has already been considered by the modeling and simulation (M&S) community as a promising approach for producing executable simulations from models [20], and MDD approaches focused particularly on ABMS have been proposed. On the one hand, there are MDD approaches that rely on—or are inspired by—Unified Modeling Language (UML) diagrams. In these approaches, effectiveness is compromised because UML is not expressive enough to specify intricate aspects of agent-based systems [9]. On the other hand, there are MDD approaches that propose new metamodels or modeling languages. However, they consider modeling and code generation of just a few aspects of agent-based simulations, leaving much left to be developed in specific applications. Sophisticated agent features which are recurrent in agent-based simulations, such as adaptation or learning, are not considered. Moreover, these MDD approaches have been mostly evaluated from the point of view of *feasibility*. Evaluation, with quantification, of *effectiveness* is almost never considered. Therefore, there is a lack of empirical evidence regarding the effectiveness that MDD approaches promote to the development 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. Previous work on MDD showed that the more specific the application domain, the higher the chance of success [46]. Therefore, we focus on the adaptive traffic signal control (ATSC) domain, in which autonomous agents are in charge of managing traffic light indicators and should be able to adapt their control policy in order to optimize traffic flow. We thus present an MDD approach for developing agent-based simulations in the ATSC domain. To develop our MDD approach, we (i) performed a domain analysis; (ii) designed a metamodel and modeling language; and (iii) developed model-to-code transformations. For the domain analysis, we considered existing simulations, which adopt either adaptation or reinforcement learning techniques in our target domain, to keep the scope limited due to the aforementioned reasons. Because these techniques are often complex to develop, the designed modeling language provides building blocks for modeling together with automated transformations for code generation. Although we focused on a specific domain, concepts included in our modeling language can potentially be adopted in other domains. In order to analyze the productivity gains of our MDD approach, we compared the amount of design and implementation artifacts produced using our approach and traditional simulation platforms. Results indicate that our approach reduces 60–86% of the workload to develop agent-based simulations in the ATSC domain.

Specifically, this work provides the following contributions: (i) a domain-specific modeling language (DSML) and code generator, which together support the development of applications in the investigated domain and potentially in similar domains; and (ii) an empirical evaluation that concretely demonstrates the benefits of MDD for developing agent-based simulations in the ATSC domain. The DSML metamodel captures aspects that have not been considered in existing metamodels for agent-based simulations and thus can be considered as a contribution to the ABMS field as well. Additionally, from the steps followed to identify recurrent concepts that are present in existing agent-based simulations, we describe a derived bottom-up domain analysis method. This method led us to successfully develop an MDD approach in the ATSC domain and can potentially be used to identify and to abstract concepts in other domains.

2. Background on 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) [78], in contrast, models are *first-class citizens*, and the development is driven by modeling artifacts [81]. 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.[1] This (semi-)automatic generation of source code improves efficiency, productivity, reliability, reusability, and interoperability. Therefore, 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 [80]. In the context of the software development in industry, Tovalnen and Kelly [84] showed that MDD approaches can increase productivity by a 5–10x factor.

According to Schmidt [78], an MDD approach combines the following elements: (i) domain-specific modeling languages, and (ii) transformation engines and generators. The former allows effectively expressing domain concepts, while the latter introduces automation. These elements are described as follows.

A domain-specific modeling language (DSML) is a modeling language designed for a particular domain, trading generality for expressiveness [81]. The type system of a DSML formalizes the application structure, behavior, and requirements of that particular domain [78]. Therefore, DSMLs provide high-level, off-the-shelf, abstractions for the business-related concepts and process. For instance, a DSML 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, DSLMs open up their application domain to a larger group of users [60]. The focus on a particular domain—in our case, ABMS—brings reading and learning efficiency and allow expressing *what*, instead of *how*, to compute [23,56].

A common way of describing DSMLs is metamodeling [78]. 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 DSML and thus must be included in the metamodel [82]. 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 shown in Fig. 1. 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.

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 [6]. 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 DSML 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 [81].

The second element of an MDD approach, model transformation, is a process that converts models. 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 statements, as shown in Fig. 1. 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.

Although there is no restriction in the way metamodels and model transformations are specified, there are initiatives toward standardization in MDD. The model-driven architecture (MDA) by the Object Management Group (OMG) [62,79] is the most prominent one. Indeed, the four abstraction layers presented in Fig. 1 are part of the MDA metamodeling framework. In this framework, the *meta object facility* (MOF) is the standard meta-metamodel, and the *query/view/transformation* (QVT) and *mof2text* languages are proposed for specifying *m2m* and *m2c* transformations, respectively. Other standards have been proposed as alternatives to MOF and QVT/mof2text, such as the EMF Ecore[2] meta-metamodel and the ATL[3] and Xpand[4] transformation languages.

[1] The term *model-based* is often used in the M&S literature to describe approaches that systematically use models as the primary artifact of a simulation development process. In this work, we adopt the term *model-driven* to refer to these approaches. Our choice is aligned both with the software engineering terminology and that used by Çetinkaya et al. [18].

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

[3] http://www.eclipse.org/atl.

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

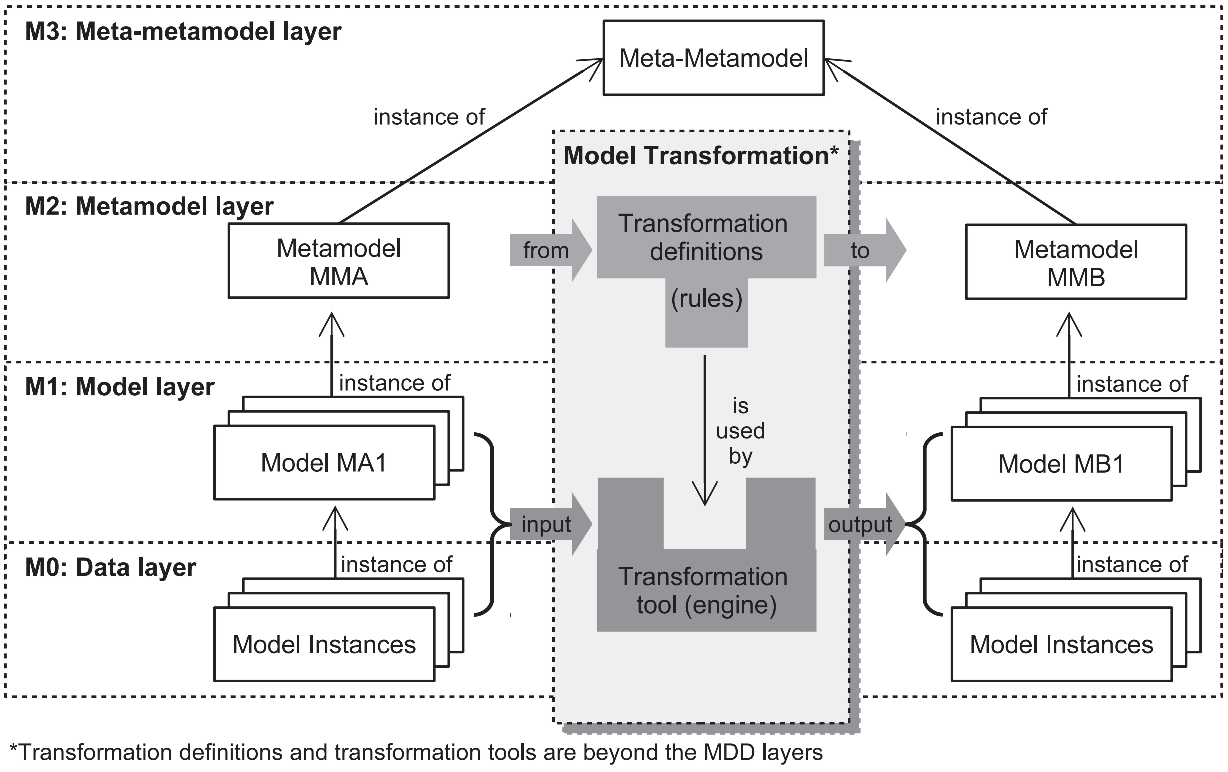


Fig. 1. MDD abstraction layers and model transformation elements. Adapted from [14].

3. MDD for ABMS

In this section, we describe the proposed model-driven development approach for agent-based simulations. We start by introducing the selected agent-based simulation domain: adaptive traffic signal control (ATSC). As previously stated, we focus specifically on this domain due to the trade-off between generality and expressiveness. Also, there is evidence that regarding MDD the more specific the application domain, the higher the chance of success [46]. After introducing the ATSC domain, we describe the conducted domain analysis, then we present the resulting metamodel, modeling language, and model-to-code transformations.

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 [10]. 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 [11]. By being endowed with learning capabilities, for example, agents can refine traffic control policies in real time [59] and thus optimize the overall traffic flow for a more efficient use of the existing infrastructure. Indeed, the successful use of agents have been reported in traffic specialized literature (e.g., [22,27,50]). For a review of agent-based systems for traffic control, we refer the reader to elsewhere [11,22]. We next describe the domain analysis conducted to identify the domain concepts present in ATSC agent-based simulations. These domain concepts are the basis for creating the metamodel for agent-based simulations.

3.1. Domain analysis

Any source of explicit or implicit domain knowledge can be considered in the domain analysis [60]. 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 *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 traffic lights [24,34] and reinforcement learning for traffic light control [59,63,90].

To guide the domain analysis, we considered existing work in this context, focused on MAS. Hassan et al. [45] proposed a process to guide the identification and formalization of domain concepts. A similar initiative was proposed by Garro and Russo [33]. 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 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 [41]. 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 was composed of the following steps.

Step 1 **Concept Preliminary List.** A list of MAS-related concepts was identified using the existing ATSC simulations (e.g., agents and the environment), following the steps of Hassan et al. [45] and Garro and Russo [33].
Step 2 **ODD-based Refinement.** The ODD protocol was 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, were analyzed in order to find their underlying essence. The analysis considered recurrent characteristics and behaviors. Similar concepts were abstracted as a single, essential concept, or generalized to a parent concept. During the analysis, a list of abstractions was built, containing the domain terminology and concepts, and generalizations.
Step 4 **Domain Modeling.** The list of abstractions was used to build the domain model, which is described in the next section.

Considering our investigated domain, after performing the step 1 described above, we identified different recurrent MAS-related concepts, which are: environment, agents and their perceptions, vehicles, and demand. They 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. Fig. 2a presents 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 shown in Figs. 2b and 2 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 associated 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 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. Fig. 2c shows 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. Wiering [90], Oliveira and Bazzan [63], and Mannion et al. [59] 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 [24,34] 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

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 metamodel was built following the abstraction step of our domain analysis method. It was constructed upon the EMF Ecore meta-metamodel, introduced in Section 2. In the following two sections we describe the elements that compose the metamodel.

3.2.1. Agent and traffic control elements

Fig. 3 presents the metamodel elements related to agents and to how traffic control elements are incorporated into them. The basic elements of the metamodel are agent, entity, and attribute. An *entity* represents any object 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

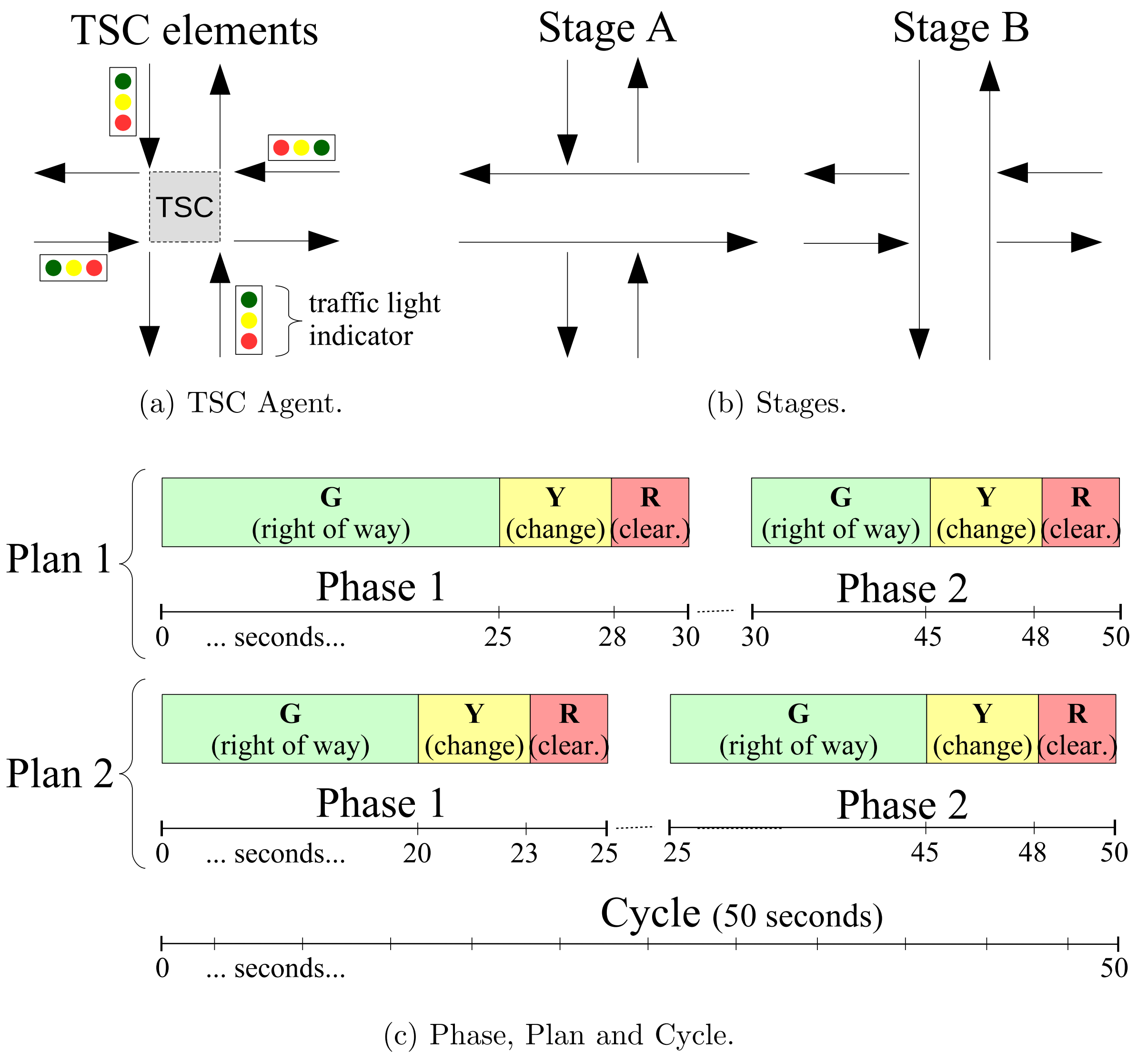


Fig. 2. ATSC domain terminology and concepts.

Bernon et al. [12]. However, given that we follow a bottom-up approach, existing metamodels are not used as a start point to avoid bias that can lead to the introduction of unnecessary or overly complex concepts. 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, essentially, a TSC is 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 to manage the known streams. 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, and abstract it as *actuator states*. Therefore, each actuator state 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. Fig. 4 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 *flow control*

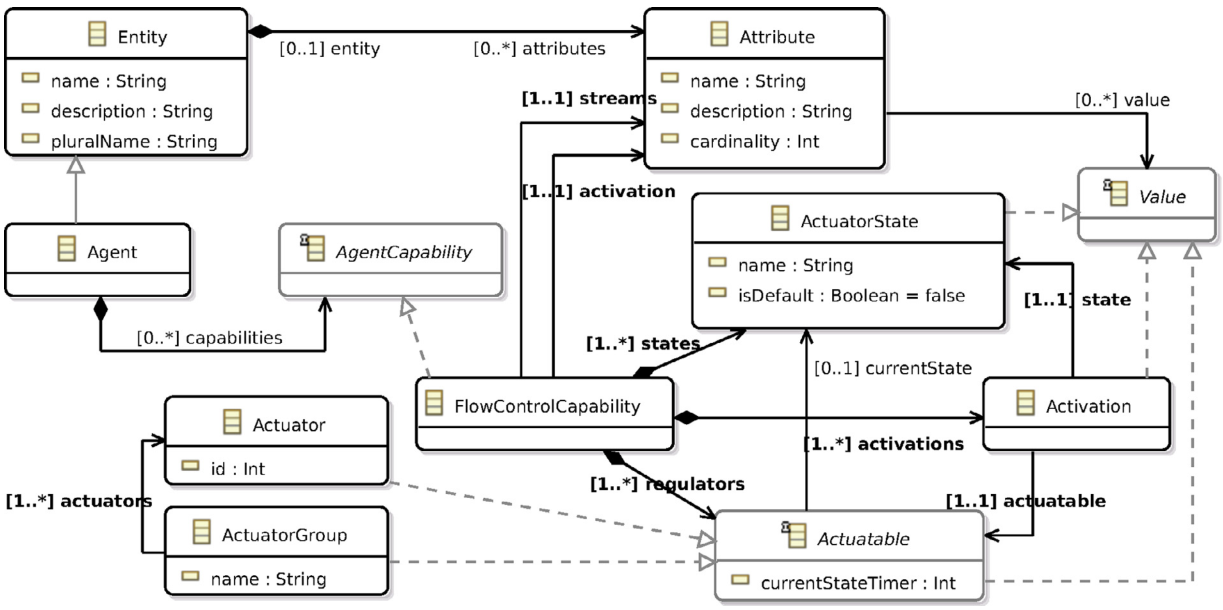


Fig. 3. ABMS metamodel (Part 1): Agents and traffic control elements.

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 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 endows the agent the ability to select an activation that is appropriate to the current traffic conditions. Next, we describe the decision capabilities that were identified in the domain analysis and incorporated into the metamodel.

3.2.2. Decision capabilities

A *decision capability* represents a decision policy. Such a policy describes how to choose one amongst many available *decision options*. Fig. 5 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 Fig. 3, 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).

Periodically, the *decide* operation of each decision capability is activated and its output updates the value of a particular agent attribute. From the domain analysis, we identified three types of decision capabilities: state machines, adaptation, and learning. A *state machine* represents a fixed decision policy. It is composed of *states*, which are pairs of decision option and transitions. A transition specifies the conditions that activate a particular *target* state. Such condition is specified with an *expression*, which can consider any agent attribute in addition to the current state of the machine and its timer. To represent state machines, we adopt a subset of the Unified Modeling Language (UML) Statemachines metamodel[5]—highlighted with gray in Fig. 5. Because we used a bottom-up approach to specify our metamodel, we left out elements that would unnecessarily increase its complexity.

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*.

A *learning* capability allows an agent to learn a decision policy. We consider a *reinforcement learning*[6] 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 *learning state definition*[6], which use expressions to describe the tuple of elements that characterize a state. The concrete states are created during the simulation execution, from a

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

[6] Elements related to reinforcement learning are highlighted in gray because they come from the existing literature in this context. Our metamodel includes them in terms of the Ecore elements.

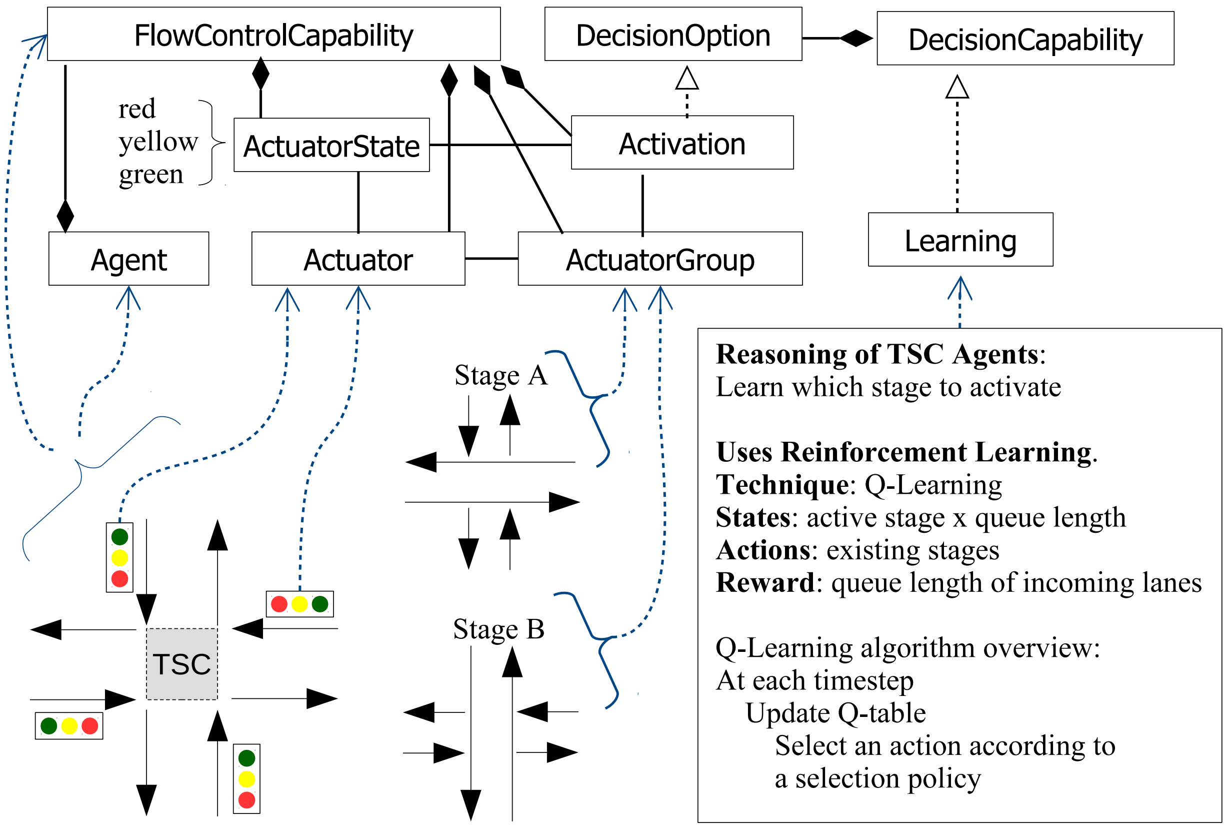

FlowControlCapability
DecisionOption
DecisionCapability
red
yellow
green
ActuatorState
Activation
Agent
Actuator
ActuatorGroup
Learning
Stage A
Stage B
TSC
Reasoning of TSC Agents: Learn which stage to activate
Uses Reinforcement Learning.
Technique: Q-Learning
States: active stage x queue length
Actions: 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. 4. Example of concept abstractions.

cartesian product of this tuple of elements. Actions are represented by the decision options that are related to the decision capability. Finally, the reward signal is specified as an *expression*. As illustration, Fig. 4 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 [88].

3.3. Language concrete syntax

In order to model agent-based simulations in an expressive way, the language should provide building blocks for ATSC elements. In the work of Santos et al. [77], a domain-specific language—DSL4ABMS—focused on modeling the simulated environment of agent-based simulations was proposed. The language adopts a graphical representation to reduce the effort required to identify model elements and their relationships, and provides building blocks for recurrent elements of the simulated environment. DSL4ABMS was empirically evaluated with humans, and results provided evidence that it decreases the time needed to understand agent-based simulations. In this work, we extend the DSL4ABMS language. The metamodel presented in the previous section was incorporated into DSL4ABMS, and additional representations were developed for its elements.

DSL4ABMS models objects that exist in the simulated environment as entities. Entities are represented using a box with at least three sections: the entity name; how it is created; and its attributes [77]. In this work, we adopted the same representation for agents. Fig. 6a presents the concrete syntax used for representing agents. Guillemots ($\ll$ $\gg$) denote placeholders for model data. The first section shows the agent name. The second section enumerates the capabilities incorporated into the agent. The last two sections describe the agent creational strategy and attributes, respectively. For details on the latter two sections, we refer the reader either to [77] or to the DSL4ABMS complete concrete syntax description available elsewhere.[7]

The concrete syntax of a *flow control capability* is shown in Fig. 6b. 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 additional attributes are created: *streams* and *activation* (previously described in Section 3.2). Other agent sections remain unaffected. Additional specifications required by a flow control capability are represented as boxes connected

[7] http://www.inf.ufrgs.br/prosoft/resources/dsl4abms/2017simpat/.

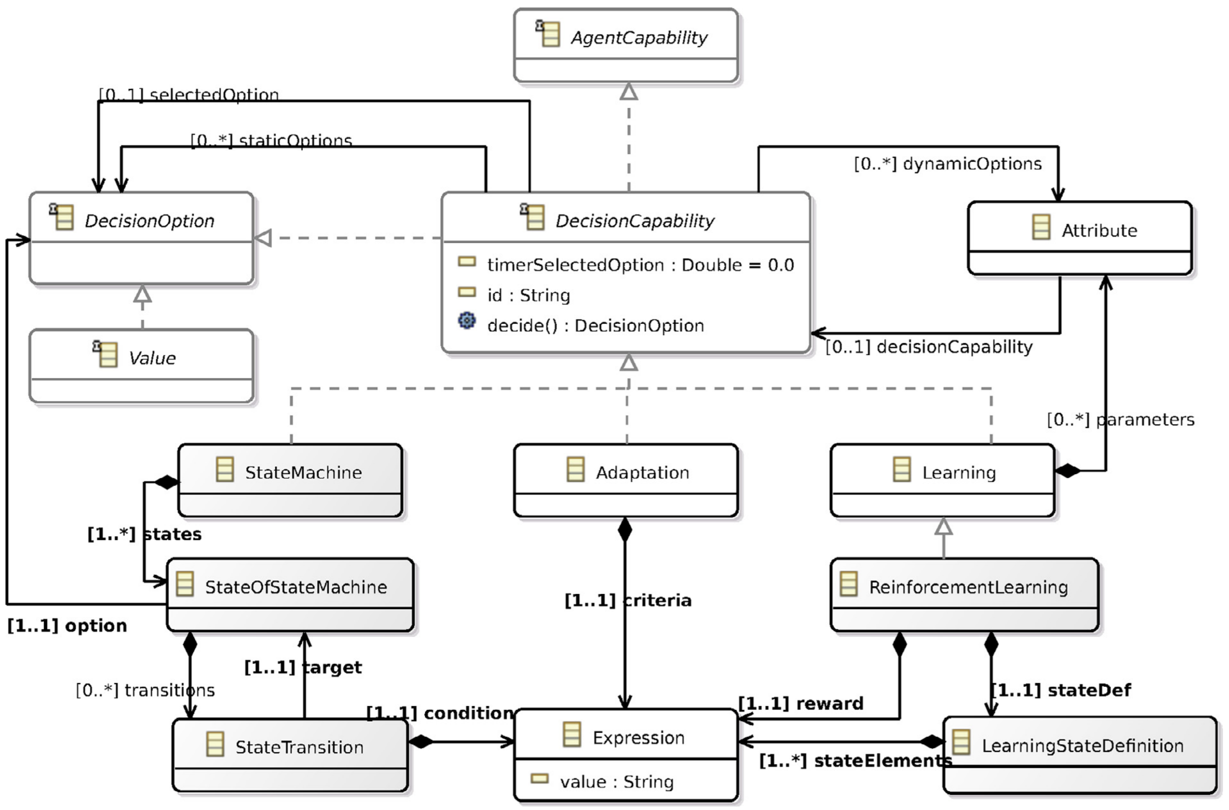


Fig. 5. ABMS metamodel (Part 2): Decision capabilities.

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.

In our extended DSL4ABMS, decision capabilities are represented as boxes whose content describes the elements required by each capability type, as shown in Figs. 7a, 7b, and 7c. The top section of each box presents 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 3.2.2). All other elements and sections are related to the specification of decision options.

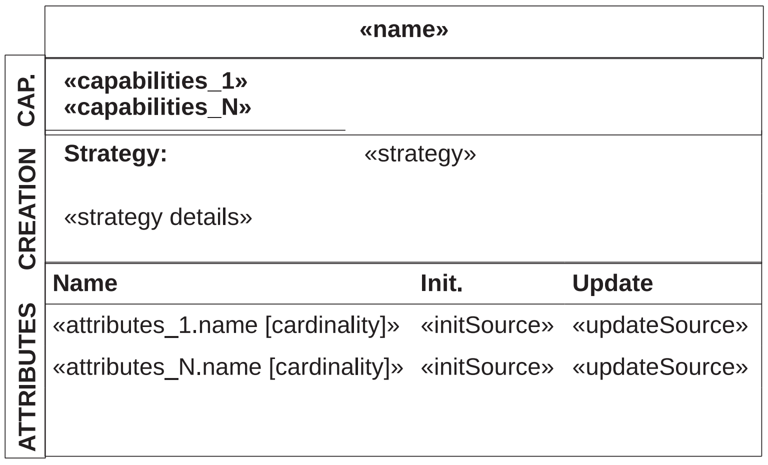
Decision options are specified with *connections*. A capability box includes an input connector (semicircle[8]), from which a connection to any model element that represents either static or dynamic decision options can be created. As mentioned in Section 3.2.2, a decision capability can also be a decision option of another decision capability. In such a case, an output connector (filled circle[8]) 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 just one connection, then all the elements of the connection target are considered decision options. For example, if the target is the *actuator(s)* element, 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. In turn, when there is more than one connection, the cartesian product of the target elements are 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 just some options should be considered, they are enumerated in this section. Consequently, the language allows constraining only static decision options. A state machine box does not require such section because the considered options are those enumerated as states of the state machine.

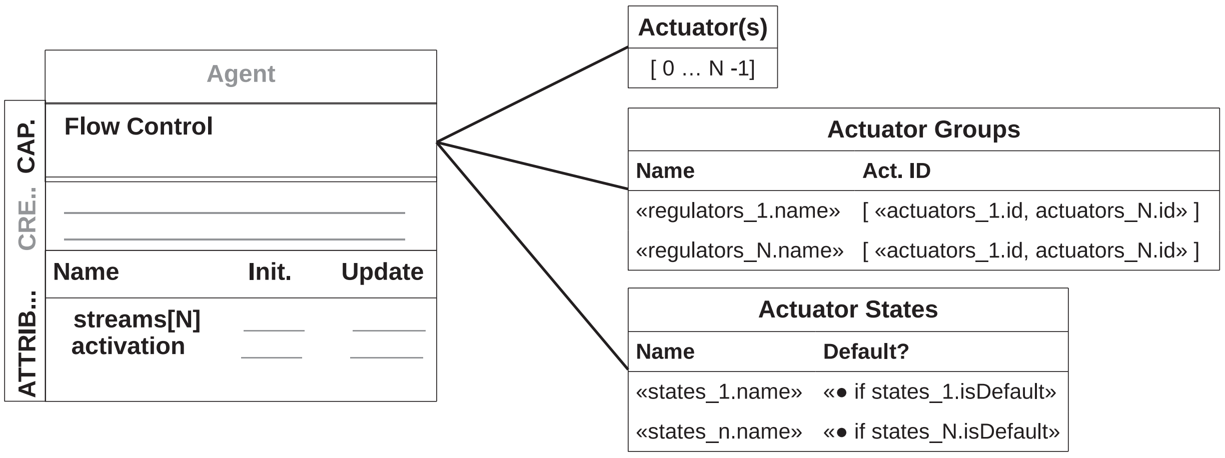
3.4. Model-to-code transformations

Our DSL4ABMS provides the support needed to model agent-based simulations. However, support for code generation is fundamental to reduce the effort to develop them. The generated code should include all the variables and operations that

[8] Semicircle and filled circle connectors were inspired by the UML component diagram.



(a) Agent.



(b) Flow Control Capability and Traffic Elements.

Fig. 6. Concrete syntax of agent and traffic related elements.

implement an agent's behaviors and capabilities, in addition to the setup and execution of a simulation, as shown in the source code template presented in Fig. 8. Elements enclosed in square brackets are only generated if specified in the model.

The agent source code must include statements for defining its attributes, which must be either designer defined or derived from agent capabilities (e.g., attributes for representing states and transitions are derived from a state machine capability). The `act` operation implements the agent behavior and it is executed at every timestep of the simulation. All other operations are related to agent capabilities. For each capability, an `initialize` operation must be generated to set it up. The *flow control capability* requires an operation to select and apply an activation (pair actuator-state), which is invoked by the `act` operation. In addition to operations that are particular to each decision strategy, a `decide` operation must be implemented for each decision capability. This operation must select and return the appropriate decision option, and it is also invoked by the `act` operation.

The simulation source code must include statements for defining the model parameters, which can be coded as attributes of the simulation. If the value of these parameters are provided by the user, operations to create input components must be generated. The `setup` operation implements the creation and initialization of entities and the environment according to the creational strategies specified in the model. Finally, the `execution` operation implements the update of entities and the environment at every timestep, in addition to the activation of agents.

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 [91], a popular agent-based

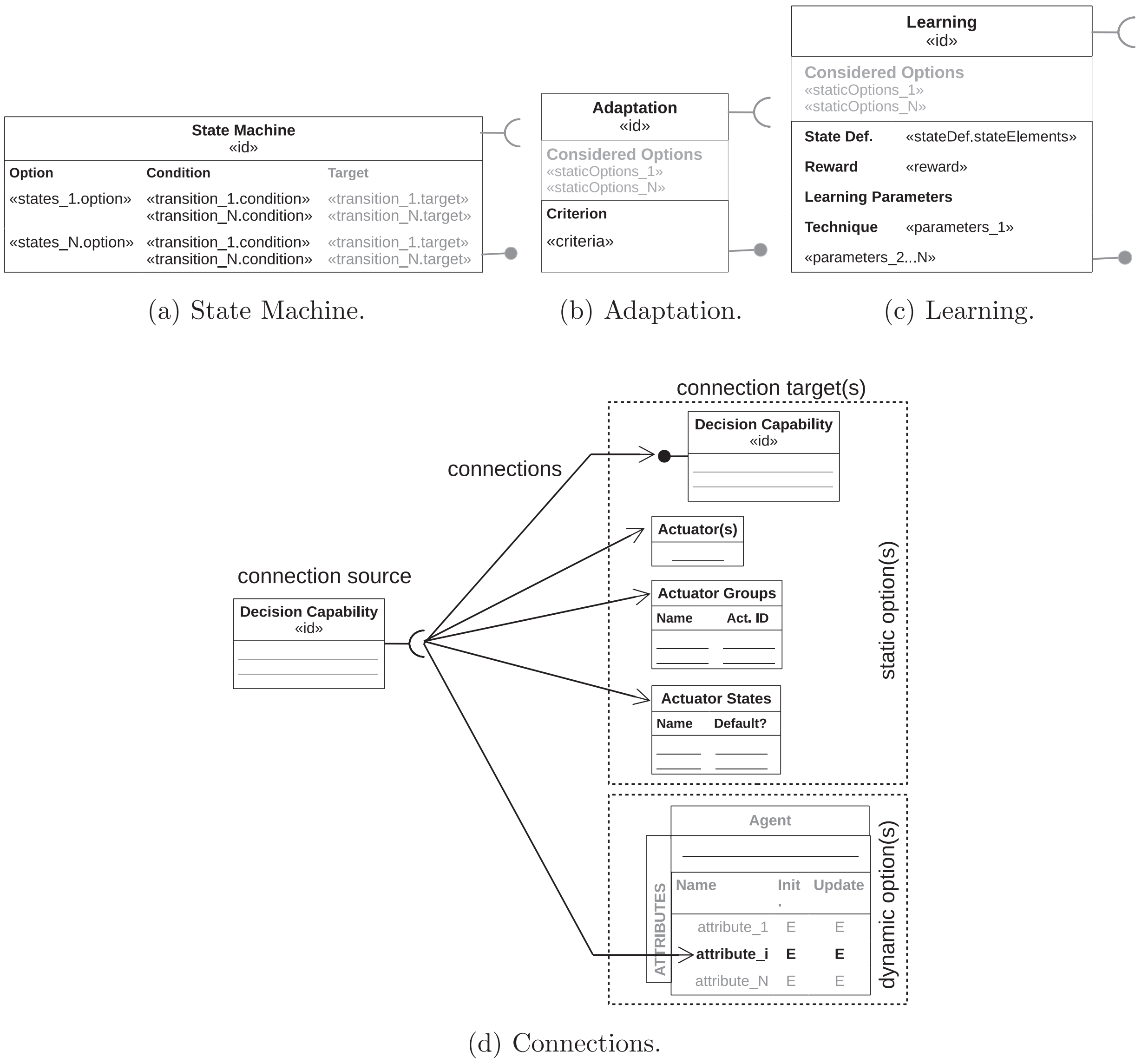


Fig. 7. Concrete syntax of decision capabilities.

simulation platform. Model-to-code transformations are performed through the use of production rules, which transform instantiated concepts of our metamodel to NetLogo code statements and blocks according to the code template previously presented. We recall that this template is platform-neutral, and thus can be used to guide the development of transformations that generate code for other simulation platforms.

In our MDD approach, production rules were specified and documented using the Xpand template language. Each Xpand template describes source code that is generated for its corresponding metamodel element. The complete Xpand specification of the production rules is available on-line.[7]

To illustrate how the developed production rules work, Table 1 describes, using natural language, a subset of the rules for transformation of agents, their attributes, and capabilities. Rules related to the reinforcement learning capability, more specifically for the Q-Learning [88] technique, are also shown. Overall, one production rule is specified for each element of the code template. 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 in 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.

```
Agent Template
  Attributes
    Designer−defined
    [Capability−derived]
  Operations
    Act
    [Flow Control
      Initialize
      Select activation]
    [Learning
      Initialize
      Compute reward
      Learn
      Decide]
    [State Machine
      Initialize
      Evaluate transition(s)
      Decide]
    [Adaptation
      Initialize
      Evaluate criterion
      Decide]
```

```
Simulation Template
  Attributes
    Model parameters
  Operations
    [Input components]
    Setup
      Create and initialize entities and the environment
    Execution
      Update entities and the environment
      Activate agents
```

Fig. 8. Code templates for agents and the simulation.

Usually, agent capabilities demand certain data structures. For example, the Q-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, which positively affects development productivity.

A verification procedure was performed to guarantee that generated simulations are adequately coded and correctly implement the expected behavior [47]. Such a verification consisted of unit tests in the different parts of our metamodel, in addition to 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.

4. Case study: Learning on traffic signal control

This section illustrates the feasibility of using our MDD approach to develop agent-based simulations in the ATSC domain. We show how agent and traffic-related elements are modeled using DSL4ABMS. To illustrate the code generation ability of our MDD approach, we show fragments of the NetLogo code produced from decision capabilities. A discussion on the effectiveness of our MDD approach is presented later, in Section 5.

The selected simulation was presented by Oliveira and Bazzan [63], who analyzed the effects that TSC agents that adapt their control policies cause in the performance of the traffic system. A cycle time of 60 seconds was considered, and three signal plans were modeled. Each plan specifies the duration of two phases: north-south and west-east. Plan 1 gives equal green phase duration time for both phases. Plans 2 and 3 give priority to the vertical or horizontal direction, respectively. In these two plans, 70% of the cycle time is allocated to the phase of the preferential direction. TSC agents use reinforcement 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, in other words, the queue of vehicles at incoming links.

Fig. 9 shows how the TSC agent is modeled with DSL4ABMS. The ABSTRACTME [61] tool was used for modeling. As can be seen, the agent is endowed with a *flow control capability* to manage the traffic flow at its intersection. The *streams* attribute has cardinality two because there are two incoming lanes: north-to-south and west-to-east, respectively. Each stream is assumed to have an actuator, whose identifiers are listed in the *actuator(s)* element. Two *actuator states* (green and red—default) are specified. Because there are only two actuators (one for each direction), no actuator groups are specified. The *activation* attribute stores the current activation—*actuator, state* pair— and it is updated periodically according to a learning capability.[9] The last two attributes are related to the agent perception and are used by the learning capability, as described later. Instances of this agent are created based on the specified creational strategy, and they are located at the traffic nodes of the traffic network.

[9] The letter E alongside an attribute denotes an expression definition. In the particular case of the *activation* attribute, the expression refers to the learning capability. In turn, the letter V denotes a static value. This notation was introduced in [77].

Table 1
Production rules (Partial View).

Production Rule	Transformation to NetLogo code
agent type	For each *Agent* $\rightarrow$ `breed`
agent attributes	For each ≪`agent.attributes`≫ $\rightarrow$ `breed-own`
	For each ≪`agent.capabilities`≫ $\rightarrow$ `breed-own` for derived attributes (e.g., Q-table)
flow control capability	For each ≪`agent.capabilities`≫ of type *FlowControlCapability* $\rightarrow$
	flow control initialize
	flow control select activation
flow control initialize	For each ≪`regulators`≫ $\rightarrow$ `set` statements for setting up actuator and groups
	For each ≪`states`≫ $\rightarrow$ `set` statements for setting up actuator states
	For each ≪`activations`≫ $\rightarrow$ `set` statements for setting up activations
flow control 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* $\rightarrow$
	qlearning initialize
	qlearning compute reward
	qlearnig learn
	qlearning decide
qlearning initialize	For each ≪`stateDef.stateElements`≫ $\rightarrow$ `set` statements for setting up states
	For each `staticOptions`≫ and ≪`dynamicOptions`≫ $\rightarrow$ `set` statements for setting up actions
qlearning compute reward	For the `reward` expression $\rightarrow$
	`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 $\text{argmax}_a Q(s, a)$ with probability $1-\epsilon$, and a random action with probability ϵ.

Signal plans are specified as *state machines* because they are fixed decision policies for selecting an activation according to the elapsed time. For example, the *plan north-south* state machine has as decision options combinations of *actuators* and *actuator states*, as can be seen from the connections. However, only two out of the four combinations constitute states of the state machine. The transition between these states are based on time: the state *(0, green)* is activated for 42 seconds (which is 70% of the cycle time) and the state *(1, green)* for 18 seconds.

Finally, there is a *learning capability* to endow the TSC agent with the ability to select plans. The state definition of Oliveira and Bazzan [63] is based on the queue length of the incoming links. Such definition resulted in three states: *equal* when the queue length is the same on both links; *north/south* when the north-south queue is longer than the other; and *west/east* when the west/east queue is longer than the other. In our model, the expression that specifies a tuple of booleans whose values define these three states is shown in Eq. (1).

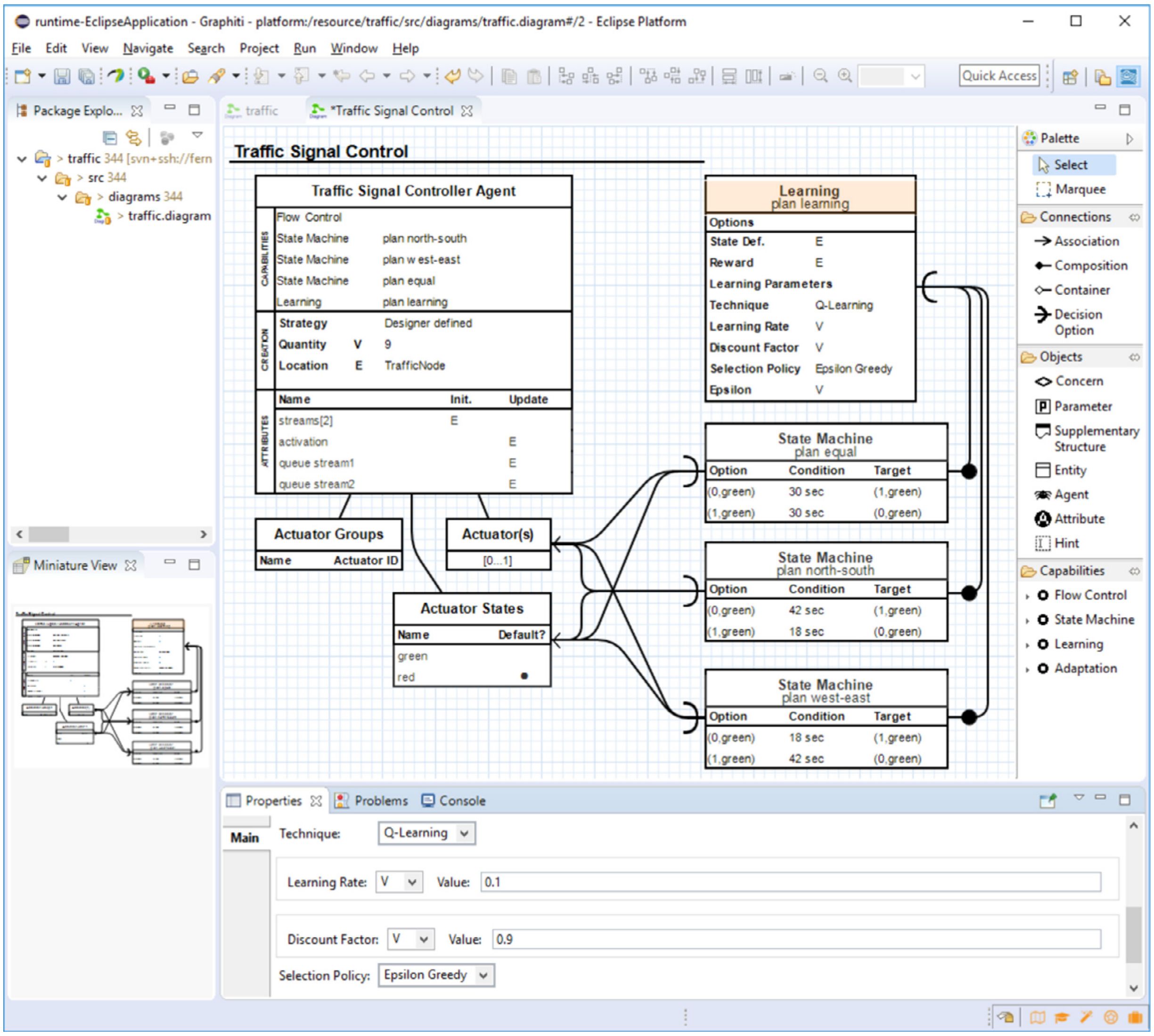
$$(queue\ stream1 = queue\ stream2), (queue\ stream1 < queue\ stream2), (queue\ stream1 > queue\ stream2) \tag{1}$$

The *decision options* of the learning capability are the three state machines that represent signal plans. The reward function proposed by Oliveira and Bazzan [63] is shown in Eq. (2). It takes into account the queue of vehicles stopped at each incoming link. This function is used as is in our model.

$$reward = \frac{1 - (2 * (queue\ stream1 + queue\ stream2) - |(queue\ stream1 - queue\ stream2)|)}{10} \tag{2}$$

The last section of the learning capability shows the learning parameters. Authors used the basic Q-Learning mechanism and experienced different values for its parameters. In the model of Fig. 9, these parameters were filled in with arbitrary values to illustrate a complete model. With the code generation ability provided by our MDD approach, a runnable simulation can be generated for any particular set of parameters in order to find the combination that leads to the best results.

As DSL4ABMS visual elements are part of diagrams to model a simulation, instances of metamodel elements are automatically created from them to represent the simulation model. The model is saved in the XML Metadata Interchange (XMI) format, a standard adopted by the EMF Ecore framework. Appendix A shows a fragment of the XMI file on which the *plan learning* and *plan north-south* capabilities are specified for the TSC agent.



Technique: Q-Learning

Learning Rate: V Value: 0.1

Discount Factor: V Value: 0.9

Selection Policy: Epsilon Greedy

Fig. 9. DSL4AMBS model of the ATSC simulation created with the ABSTRACTME Tool.

NetLogo code is automatically generated by our code generator, which reads the XMI file and applies the model-to-code transformations previously described. The generated code includes all the required statements to set up and run TSC agents. To illustrate the output of the transformation rules, Appendix B presents fragments of the NetLogo source code that were generated from the modeled simulation, in particular for setting up and running a simulation, and for the state machine *plan north-south* and the reinforcement learning *plan learning* capabilities.

Based on this case study, we demonstrate the feasibility of using our MDD approach to model agent-based simulations in the ATSC domain and have its NetLogo source code produced automatically. The generated code contains routines that implement both flow control and decision capabilities of TSC agents, and it is ready to be executed in the NetLogo simulation platform. However, 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 design such a simulation model is similar to or higher than the effort to develop it directly in the chosen simulation platform. In next section, we evaluate the effectiveness of our MDD approach for reducing the development effort and thus for increasing productivity in ABMS.

5. Evaluation

MDD has been effective for increasing productivity in software development by focusing on both expressive modeling of domain concerns and automated code generation. In this section, we present an evaluation of the effectiveness of our MDD approach. The evaluation consisted of an empirical study on which a software engineering metric was used to assess the productivity of using our MDD approach to develop agent-based simulations.

5.1. Goal and procedure

The goal of our empirical study is to assess the productivity gains promoted by our MDD approach. The following procedure was adopted: i) we selected a set of existing agent-based simulations in the ATSC domain; ii) we modeled these simulations using DSL4ABMS and generated its corresponding source code using our transformation engine; and iii) we used a metric of productivity to compare models created using DSL4ABMS with existing simulation models.

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 [3], COCOMO 2.0 [17], SEER-SEM [31,49], and SLIM [70] rely on size metrics to estimate the required effort as person-months or calendar months. Even recent estimation models focused on web applications have considered size metrics as input [5]. Indeed, previous work showed that system size is the most significant factor that affects development effort, quality and construction time [2].

Therefore, our evaluation considered the size of simulation models to measure the effort required to produce them. More specifically, we used *lines of code* (LoC), which is frequently used as a software size metric [17,31,49,70]. In fact, LoC is a metric that has been used to evaluate and compare design and development implementation effort in MDD approaches [64]. This is also the case in multiagent systems [21]. 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. Given that our focus is on TSC agents, our study considered only code dedicated to develop them and their associated behaviors. Comments and code block delimiters were ignored.

DSL4ABMS uses a graphical representation. Thus, for a fair comparison, we used the *atomic model element* (AME) metric [13] to measure the design development effort of DSL4ABMS simulation models. An AME is a visual modeling element that is equivalent to a LoC. To classify a modeling element of DSL4ABMS as an AME, we adapted a generous estimation rule proposed for UML class diagrams [13]. 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. Target simulations

In the domain analysis activity, we identified three adaptive strategies recurrently used in ATSC agent-based simulations: adaptation, learning, and state machines. In order to evaluate the effectiveness of our MDD approach for modeling these strategies, we selected one existing simulation for each of them. As discussed, existing MDD alternatives cover only conceptual ABMS-modeling and, consequently, existing simulations are usually 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. Moreover, as we adopted software size metrics, 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 MDD 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 [92] for evaluating the effectiveness of modeling them as state machine capabilities. The simulation of self-organizing traffic lights [34] was selected to evaluate the *adaptation* capability. Both simulations are available for NetLogo[10,11]. Finally, to evaluate the effectiveness of using learning for traffic light control, we used the simulation of TSC agents with reinforcement learning [63] previously presented in the case study section. This simulation is available for the ITSUMO simulation platform[12].

5.3. Results

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) **Development Effort**: the sum of AMEs and MLoCs, which is the (design and implementation) development effort of the manually created portion of the simulation model; and (iv) **GLoCs**: the number of lines of code generated from AMEs.

As can be seen, our DSL4ABMS 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. From the complete source code required to run a simulation, approximately 50% is automatically generated. Vehicle and visualization-related codes, which are out of the scope of our approach, corresponds to the remaining 50%.

[10] http://www.ccl.northwestern.edu/netlogo/models/TrafficGrid.
[11] http://www.turing.iimas.unam.mx/~cgg/sos/SOTL/SOTL.html.
[12] http://www.inf.ufrgs.br/maslab/traffic/itsumo/.

Table 2
Size comparison.

Simulation	AMEs	MLoCs	Development Effort[a]	GLoCs
State Machine / Fixed Plan				
NetLogo	1	34	35	1
Our approach	14	0	**14**	116
Adaptation / Self-organizing Traffic Lights				
NetLogo	1	82	83	1
Our approach	15	0	**15**	103
Learning / Reinforcement Learning				
ITSUMO	0	257	257	0
Our approach	38	0	**38**	298

[a] Development Effort = AMEs + MLoCs

It is also possible to observe that the total amount of lines of code (MLoCs + GLoCs) produced by our MDD approach for TSC agents and their behavior is greater than the total amount of lines of code in the existing implementation, for all simulations. Given that our 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, GLoCs are not considered as evidence of development effort, because no human effort is required to produce them.

The presented results gives evidence that our MDD approach is effective for reducing the effort required for developing agent-based simulations because the effort to design models using DSL4ABMS is lower than the effort to implement code using NetLogo or ITSUMO. Given that an AME is equivalent to an MLoC, the sum of AMEs and MLoCs produced when using our MDD approach is always lower than when using the simulation platforms. Although the number of AMEs produced using DSL4ABMS is the highest, it is lower than the number of MLoCs produced using the simulation platforms, leading to a reduction in the combined design and implementation development effort. Furthermore, the rule adopted for counting AMEs led to a fine-grained and platform independent evaluation. Therefore, a few elements of DSL4ABMS 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 a stronger confidence of that our approach *reduces the human effort* required to develop agent-based simulations in our domain.

In summary, results of the empirical study showed evidence that our MDD approach leads to productivity gains. These results, combined with the case study presented in Section 4, allow us to conclude that our MDD approach is *feasible* and *effective* for developing signal control agents in the domain of ATSC simulations.

5.4. Threats to validity

In this section, we discuss threats to study validity. An internal threat is that the AME metric is based on an estimation rule to classify elements of DSL4AMBS as AMEs. As said, we adopted a generous estimation rule [13]. In spite of a few elements of DSL4AMBS 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 our MDD approach and its modeling language. In fact, scalability can be a problem in languages with graphical notation. We mitigated this problem in the DSL4ABMS language by introducing the notion of *concerns*. Simulation concepts can be split into different diagrams that represent different concerns [77]. Indeed, the partitioning of the model is recommended when engineering DSMLs in order to keep models manageable [85]. In this work, we focused exclusively on ATSC simulations in which agents in charge of managing traffic signal controls are the main aspect considered. Therefore, our work deals exclusively with the concern of traffic signal controller agents. A future work could be to analyze and incorporate other aspects into our DSML, such as microscopic vehicle models (e.g., car-following and lane change models). These aspects would be modeled as additional concerns with their own diagrams (e.g., vehicles concern), and thus would not compromise the scalability of our modeling language and the validity of our results. It is important to notice that our evaluation has not raised evidence against the use of MDD in the ATSC or others domains.

The modeling of one simulation for each decision capability is another external threat to validity. Additional simulations were enumerated when we planned our experiment, either related to learning ([1,59,73]) or adaptation ([35,36]). 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 et al. [59] and Oliveira and Bazzan [63] 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 ATSC simulations that adopt one of the three adaptive strategies considered in this work, thus mitigating the threat regarding our selected simulations. Future work could incorporate other strategies into our MDD approach, which have been reported in the literature, such as *auctions* [71], without invalidating the results presented in this work.

6. Related work

In this section, we discuss work related to our MDD approach. We start reviewing applications of MDD in the broad M&S field, then we discuss MDD approaches targeted to MAS, and finally we narrow the review to approaches targeted to ABMS. We conclude this section with a discussion on the effectiveness evaluation in these related approaches.

6.1. MDD and the M&S field

MDD has already been considered by the M&S community as a viable approach for producing executable simulations from models. Models specified using either the Business Process Model and Notation (BPMN) or the DEVS Modeling Language (DEVSML) were considered by Çetinkaya and coleagues [18,19]. In their work, model transformations were proposed for generating simulations targeted to the Discrete Event Systems (DEVS) paradigm. Bocciarelli and D'Ambrogio also considered BPMN models [16], in addition to models specified using the Systems Modelling Language (SysML) [14], but proposed transformations for generating simulations targeted to the Distributed Simulation (DS) paradigm. In contrast, our MDD approach is targeted to the ABMS paradigm, in which simulations are specified considering agents, interactions, and the environment. Modeling agent systems using languages conceived for modeling systems in paradigms other than ABMS would potentially raise expressiveness issues, similarly to what was previously reported with respect to using UML for modeling agent-systems [9]. Our DSL4ABMS modeling language, instead, provides elements for modeling recurrent aspects of agent-based simulations, such as flow control and learning capabilities, which reduce the abstraction gap and thus improve expressiveness and productivity.

A formal MDD framework for M&S—MDD4MS—has already been proposed [20]. The framework enumerates and formalizes the elements required by an MDD approach for M&S in order to support modeling and code generation. Metamodels, modeling languages, models, model transformations, and the final simulation source code are enumerated as the main elements. Additional elements are the metamodeling and transformation languages, and the programming language for final code generation. Although our MDD approach was not described using the formal terminology of Çetinkaya et al. [20], it provides the elements enumerated in the MDD4MS framework and thus can be considered an MDD approach, but targeted to the ABMS simulation paradigm. Because the goal of our MDD approach is to provide a practical and effective way for modeling simulations and have their source code generated automatically, it is described using the terminology adopted in a software engineering systematic method for developing domain-specific languages [82]. This method lays solely on standards and frameworks recurrently used in industry (e.g., MOF, XPand) to describe an MDD approach.

6.2. MDD and MAS

With respect to model-driven development of agent and multiagent systems in general, Bauer and Odel [9] investigated the use of UML for modeling agent-systems. In spite of being a modeling language widely adopted in industry to specify software systems, the authors concluded that UML—and even its extension Agent UML [8]— is not expressive enough to specify intricate aspects of agent-based systems. For example, off-the-shelf constructs to express aspects such as reproduction and emergent phenomena are missing. According to the authors, a useful way for representing such aspects, at a higher level of abstraction, should be developed so as to model agent systems effectively.

Kardas [52] reviewed a selection of model-driven approaches for MAS and classified them into three categories: i) complete MDD approaches with multiple metamodels at distinct abstraction levels and model transformations—Cougaar MDA [40], PIM4Agents and its DSML4MAS language [42,43,86,87], and MDD4SW Agents [53]; ii) partial MDD approaches with single metamodels and either model-to-model or model-to-code transformations—Malaca [4], CAFnE [48], TAOM4e [69], PIM4SOA [94], AMDA [93]; and iii) extensions of existing MAS methodologies to support MDD—INGENIAS Development Toolkit (IDK) [67], Model-driven Tropos [68], and Model-driven ADELFE [74]. Although these MDD alternatives provide support for modeling and code generation, the author noticed that in most situations, such 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 approaches are limited because the amount and quality of the automatically generated MAS components appear to be insufficient. Furthermore, it is important to notice that these alternatives are focused on multiagent models, and thus simulation aspects are uncovered.

6.3. MDD and ABMS

Many existing MDD approaches targeted to ABMS rely on—or are inspired by—UML diagrams for modeling simulations. The model-driven approach of Iba et al. [47] 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 language for creating simulation models on the basis of the proposed metamodel. While behaviors are specified with UML activity, communication, and statechart diagrams, all other elements are specified with UML class diagrams.

Duarte and DeLara [26] proposed a model-driven approach that provides a modeling language (ODiM) and model transformations to generate code to the MASON simulation platform. In the ODiM language models are specified by means of four diagrams: agent types, agent's behaviors, agent's sensors and actuators, and initial configuration. The concrete syntax of the ODiM language resembles UML with respect to the modeling of agent's behaviors (UML statecharts) and agent types, sensors and actuators (UML class/object diagrams).

Garro et al. [32] 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 certain elements of these diagrams into code artifacts.

As reported by Bauer and Odell [9], 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. Therefore, effectiveness is compromised in these UML-based related work. In turn, our MDD approach provides a domain-specific modeling language with off-the-shelf elements for specifying these recurrent, high-level aspects of agent-based simulations in models.

Alternatively, there are MDD approaches for ABMS 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 [55] 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. Neither model transformations nor code generation is provided.

The MAIA metamodel [37] captures social concepts such as norms and roles. It is based on institution analysis and therefore is particularly focused on agent-based social simulations. In a later work [38], 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 perform transformation of aspects such as the simulation setup.

Ribino et al. [72] analyzed existing agent-based simulations in order to retrieve common and specific elements. A conceptual metamodel was proposed, highlighting the main elements and relationships found in the considered simulations. According to the authors, such metamodel acts as a concept repository and thus may be used as a guideline for designing new simulations. Model-driven development is not a concern in this work. Therefore, the feasibility of producing ready-to-run simulations from these conceptual models is not considered.

In the IODA methodology for agent simulations [57], 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 what interactions an agent may perform or undergo, and with what other agents. However, there are no available coarse-grained simulation building blocks and thus complex interactions must be specified from scratch.

The use of the INGENIAS [66] modeling language to specify agent-based simulation models was proposed by Fuentes-Fernandes et al. [29]. The INGENIAS language provides elements for modeling agent-related aspects such as agents, roles, goals, tasks, events, interactions, and societies. Similarly, Sansores and Pavón [75] 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 [76] and the scheduling and management of agents' life-cycle [39]. 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 [52], the INGENIAS development kit has issues with respect to code generation—code is generated only at the template level.

MDD alternatives in the context of traffic control simulation were already considered. D'Ambrogio et al. [25] adopted a matrix representation for modeling traffic intersections and crossing trajectories (stages) with their green, yellow, and red intervals. Model transformations were proposed to transform such a traffic intersection model into a queueing-based model, instead of an agent-based model. As previously discussed in the paper, agent-based systems might be a solution to overcome the challenges of complex traffic demands. There are work in traffic specialized literature reporting the use of agents (e.g., [22,27,50]), and there are initiatives to integrate agents into existing traffic simulators (e.g., [7,83])—but integration relies on programming libraries and so on programming skills. An MDD alternative specifically targeted to agent-based traffic simulations was proposed by Fernandes-Isabel and Fuentes-Fernandes [28]. However, their metamodel is based on general, agent-related concepts such as tasks, goals, and facts. Several elements are required for specifying sophisticated agent behaviors such as a learning capability.

As can be seen, these approaches have limited support for modeling intricate aspects of agent-based simulation and for generating code from them. In our MDD approach, such sophisticated structures were abstracted and incorporated in the metamodel, and our DSML provide ready-to-use building blocks for them, reducing the burden of simulation development.

6.4. Remarks on the effectiveness evaluation

Finally, it is noteworthy to discuss how these MDD approaches have been evaluated. As previously mentioned in the background section, MDD is effective for increasing productivity in software development if the effort is focused on modeling high-level domain concerns instead of programming statements [80,84]. Such a gain in productivity would not be possible if the effort to create models using the provided modeling languages and metamodels were greater than using other ways (e.g., programming languages). Therefore, in order to show that any MDD approach is *effective* and thus improves productivity, one must go beyond showing that it is *feasible*. Only few existing approaches provided empirical evidence of the effectiveness of MDD approaches either in the modeling and simulation domain [15] and, more broadly, in software engineering [21,51].

With respect to related work on MDD targeted ABMS, 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 the simulation directly on the target simulation environment. Differently, in our work, an empirical study was conducted to provide such an evidence of effectiveness. We showed that the human effort required to create a simulation model using our modeling language is lower than using existing simulation platforms.

7. Conclusion

Developing agent-based simulations is a challenging task, because of a heterogeneous group of involved roles. MDD 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 lacks empirical evidence of its promoted benefits.

In this paper, we explored the use of this approach in the ABMS context, 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 and domain-specific modeling language, in addition to code generation for agent-based simulations in this investigated domain. The work is grounded 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, such as the flow regulator agent and decision capabilities. Nevertheless, these concepts 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 the design and implementation effort—measured in terms of size—required to develop simulation models using our domain-specific language is 60–86% lower than models created using existing simulation platforms. The number of lines of code and model elements, which are size metrics, have been used to evaluate the reduction of effort provided by MDD approaches.

Given that our language metamodel was built through a domain analysis activity that considered existing agent-based simulations, these results also provide evidence that the steps of the conducted domain analysis are effective and can potentially be used to identify and abstract concepts in other domains. This leads to a derived bottom-up domain analysis method, with which it is possible to identify concepts that are recurrent in agent-based simulations in other domains and provide ready-to-use building blocks for them.

Results presented in this paper pave the road towards an effective use of MDD for developing agent-based simulations. 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. 311558/2013-5, 442582/2014-5, and 303232/2015-3, CAPES 7619-15-4, and Alexander von Humboldt BRA 1184533 HFSTCAPES-P.

Appendix A. XMI Listings

Fig. A.1

```
<org.mdd4abms.mm:AgentBasedSimulation>
 <concerns title="Traffic Signal Control">
  <entities xsi:type="org.mdd4abms.mm:Agent" name="TrafficSignalControllerAgent"
        pluralName="TrafficSignalControllerAgents">
   <attributes name="activation"cardinality="1"
         decisionCapability="//@concerns.0/@entities.0/@capabilities.1">
    <updatePeriodicity xsi:type="org.mdd4abms.mm:ExpressionSource">
     <expression value="60"/>
    </updatePeriodicity>
   </attributes>
   <attributes name="queue_stream1" cardinality="1">
    <updateSource xsi:type="org.mdd4abms.mm:ExpressionSource">
     <expression value="| v in Vehicles : location=streams [0] |"/>
    </updateSource>
   </attributes>
   <!-- Other attributes and creational strategy, were ommited -->
   <capabilities xsi:type="org.mdd4abms.mm:FlowControlCapability">
    <states name="green"/>
    <states name="red" isDefault="true"/>
    <regulators xsi:type="org.mdd4abms.mm:Actuator" id="0"/>
    <regulators xsi:type="org.mdd4abms.mm:Actuator" id="1"/>
    <activations state="//@concerns.0/@entities.0/@capabilities.0/@states.0"
          actuatable="//@concerns.0/@entities.0/@capabilities.0/@regulators.0"/>
    <activations state="//@concerns.0/@entities.0/@capabilities.0/@states.1"
          actuatable="//@concerns.0/@entities.0/@capabilities.0/@regulators.0"/>
    <activations state="//@concerns.0/@entities.0/@capabilities.0/@states.0"
          actuatable="//@concerns.0/@entities.0/@capabilities.0/@regulators.1"/>
    <activations state="//@concerns.0/@entities.0/@capabilities.0/@states.1"
          actuatable="//@concerns.0/@entities.0/@capabilities.0/@regulators.1"/>
   </capabilities>
   <capabilities xsi:type="org.mdd4abms.mm:ReinforcementLearning"
         staticOptions="//@concerns.0/@entities.0/@capabilities.4
         //@concerns.0/@entities.0/@capabilities.2 //@concerns.0/@entities.0/@capabilities.3"
         id="plan_learning">
    <stateDef>
     <stateElements value="(queue_stream1 = queue_stream2)"/>
     <stateElements value="(queue_stream1 < queue_stream2)"/>
     <stateElements value="(queue_stream1 > queue_stream2)"/>
    </stateDef>
    <reward value="1 − ( 2 ∗ ( queue_stream1 + queue_streams ) − | queue_stream1 −
          queue_stream2 | ) / 10"/>
    <parameters name="technique" description="Technique" cardinality="1">
     <initSource xsi:type="org.mdd4abms.mm:StaticValueSource" value="Q−Learning"/>
    </parameters>
    <!-- Other learning parameters were ommited -->
   </capabilities>
   <capabilities xsi:type="org.mdd4abms.mm:StateMachine" id="plan_north_south">
    <states option="//@concerns.0/@entities.0/@capabilities.0/@activations.0">
     <transitions target="//@concerns.0/@entities.0/@capabilities.2/@states.1">
      <condition value="42"/>
```

Fig. A.1. A Fragment of the XMI file with the simulation model is shown in Fig. A.1 and Fig. A.2.

```
      </transitions>
     </states>
     <states option="//@concerns.0/@entities.0/@capabilities.0/@activations.2">
      <transitions target="//@concerns.0/@entities.0/@capabilities.2/@states.0">
       <condition value="18"/>
      </transitions>
     </states>
    </capabilities>
    <!-- Other capabilities were ommited -->
   </entities>
  </concerns>
</org.mdd4abms.mm:AgentBasedSimulation>
```

Fig. A.1. Continued

Appendix B. NetLogo Listings

B1. Source code for setting up and running a simulation

Fig. B.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 for updating agent attributes and for activating the learning capability.

```
to setup
   clear-all
   createAgents
   initializeAgentBreeds
   initializeCapabilities
   reset-ticks
end

to go
   updateAgentBreeds
   ask TrafficSignalControlAgents [
      ifelse (ticks mod 60) = 0 [
         set activation rl-plan_learning-decide
      ][
        set activation rl-plan_learning-current-decision
      ]
      TrafficSignalControlAgents-act
   ]
   tick
end
```

Fig. B.1. NetLogo generated routines to setup and run the simulation.

```
to-report sm-plan_north_south-decide
  let next_state nobody
  ; finding the transition that is triggered in the current state and its target state
  if sm_plan_north_south_0_green = sm_plan_north_south_selected_option [
    set next_state eval-transitions sm_plan_north_south_0_green_transitions
  ]
  if sm_plan_north_south_1_green = sm_plan_north_south_selected_option [
    set next_state eval-transitions sm_plan_north_south_1_green_transitions
  ]
  ; if any transition held, update the current option and its timer
  ifelse next_state != nobody[
    set sm_plan_north_south_timer_selected_option 1 ; restart at 1 since the current
          timestep should be considered as one actuation.
    set sm_plan_north_south_selected_option next_state
  ][
    set sm_plan_north_south_timer_selected_option
          sm_plan_north_south_timer_selected_option + 1
  ]
  ; reporting the current option
  report sm_plan_north_south_selected_option
end
```

Fig. B.2. Fragment of the generated code for the *plan north-south* state machine.

B2. Source code for the state machine plan North–South

Fig. B.2 shows a fragment of the code generated for the state machine *plan north-south*. 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.

B3. Source code for the learning capability plan learning

Fig. B.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 can be seen, the decision takes into account the computed reward and the current state of the agent. 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.

```
to-report rl-plan_learning-decide
   ; 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
  ; updating the current action and its timer
  ifelse next_decision != rl_plan_learning_selected_option [
     set rl_plan_learning_timer_selected_option 1
     set rl_plan_learning_selected_option next_decision
  ][
     set rl_plan_learning_timer_selected_option rl_plan_learning_timer_selected_option + 1
  ]
  ; reporting the action; it is a state machine so runresult is used to activate it
  report runresult rl_plan_learning_selected_option
end
```

Fig. B.3. Fragment of the generated code for the learning capability *plan learning*.

References

[1] M. Abdoos, N. Mozayani, A.L.C. Bazzan, Traffic light control in non-stationary environments based on multi agent q-learning, in: 2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC), 2011, pp. 1580–1585, doi:10.1109/ITSC.2011.6083114.

[2] M. Agrawal, K. Chari, Software effort, quality, and cycle time: a study of CMM level 5 projects, IEEE Trans. Softw. Eng. 33 (2007), doi:10.1109/TSE.2007.29.

[3] A.J. Albrecht, Measuring application development productivity, in: Proceedings of the joint SHARE/GUIDE/IBM application development symposium, volume 10, 1979, pp. 83–92.

[4] M. Amor, L. Fuentes, A. Vallecillo, Bridging the gap between agent-oriented design and implementation using MDA, in: Proceedings of the International Workshop on Agent-Oriented Software Engineering (AOSE 2004), Springer, New York, USA, 2004, pp. 93–108, doi:10.1007/978-3-540-30578-1_7. Notes in Computer Science.

[5] S.A. ao, J. Gómez, E. Insfran, Validating a size measure for effort estimation in model-driven web development, Inf. Sci. (Ny) 180 (2010) 3932–3954, doi:10.1016/j.ins.2010.05.031.

[6] C. Atkinson, T. Kühne, Model-driven development: a metamodeling foundation, IEEE Softw. 20 (2003) 36–41.

[7] T. Azevedo, P.J.M. de Araújo, R.J.F. Rossetti, A.P.C. Rocha, JADE, TraSMAPI and SUMO: A tool-chain for simulating traffic light control, in: 8th International Workshop on Agents in Traffic and Transportation, 2014, pp. 8–15, doi:10.13140/2.1.2739.4886.

[8] B. Bauer, J. Müller, J. Odell, Agent UML: a formalism for specifying multiagent software systems, Int. J. Software Eng. Knowl. Eng. 11 (2001) 207–230, doi:10.1142/S0218194001000517.

[9] B. Bauer, J. Odell, UML 2.0 And agents: how to build agent-based systems with the new UML standard, Eng. Appl. Artif. Intell. 18 (2005) 141–157, doi:10.1016/j.engappai.2004.11.016.

[10] A.L.C. Bazzan, Opportunities for multiagent systems and multiagent reinforcement learning in traffic control, Autonomous Agents Multiagent Syst. 18 (2009) 342–375, doi:10.1007/s10458-008-9062-9.

[11] A.L.C. Bazzan, F. Klügl, A review on agent-based technology for traffic and transportation, Knowl. Eng. Rev. FirstView (2013) 1–29, doi:10.1017/S0269888913000118.

[12] C. Bernon, M. Cossentino, M.-P. Gleizes, P. Turci, F. Zambonelli, A study of some multi-agent meta-models, in: International Workshop on Agent-Oriented Software Engineering, Springer, New York, USA, 2004, pp. 62–77. Lecture Notes in Computer Science. 10.1007/978-3-540-30578-1_5.

[13] J. Bettin, Measuring the potential of domain-specific modeling techniques, in: Second Domain-Specific Modelling Languages Workshop (OOPSLA), Seattle, Washington, 2002, pp. 39–44.
[14] P. Bocciarelli, A. D'Ambrogio, Model-driven method to enable simulation-based analysis of complex systems, in: D. Gianni, A. D'Ambrogio, A. Tolk (Eds.), Modeling and Simulation-Based Systems Engineering Handbook, CRC Press, 2014, pp. 119–148.
[15] P. Bocciarelli, A. D'Ambrogio, Modeling-to-simulation: Transformation approaches to boost automation in modeling & simulation, in: Proceedings of the Summer Computer Simulation Conference, Society for Computer Simulation International, San Diego, CA, USA, 2016, pp. 1–8.
[16] P. Bocciarelli, A. Pieroni, D. Gianni, A. D'Ambrogio, A model-driven method for building distributed simulation systems from business process models, in: Proceedings of the Winter Simulation Conference, Winter Simulation Conference, 2012, pp. 1–12.
[17] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, R. Selby, Cost models for future software life cycle processes: COCOMO 2.0, Ann. Softw. Eng. 1 (1995) 57–94, doi:10.1007/BF02249046.
[18] D. Çetinkaya, S. Mittal, A. Verbraeck, M.D. Seck, Model-driven engineering and its application in modeling and simulation, in: S. Mittal, J.L. Risco-Martín (Eds.), Netcentric System of Systems Engineering with DEVS Unified Process, CRC Press, 2013, pp. 221–248.
[19] D. Çetinkaya, A. Verbraeck, M.D. Seck, BPMN to DEVS: Application of MDD4MS framework in discrete event simulation, in: S. Mittal, J.L. Risco-Martín (Eds.), Netcentric System of Systems Engineering with DEVS Unified Process, CRC Press, 2013, pp. 609–636.
[20] D. Çetinkaya, A. Verbraeck, M.D. Seck, Model continuity in discrete event simulation: A framework for model-driven development of simulation models, ACM Trans. Model. Comput. Simul. 25 (2015) 17:1–17:24, doi:10.1145/2699714.
[21] M. Challenger, G. Kardas, B. Tekinerdogan, A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems, Softw. Qual. J. (2015) 1–41, doi:10.1007/s11219-015-9291-5.
[22] B. Chen, H.H. Cheng, A review of the applications of agent technology in traffic and transportation systems, IEEE Trans. Intell. Transp. Syst. 11 (2010) 485–497, doi:10.1109/TITS.2010.2048313.
[23] C. Consel, F. Latry, L. Réveillére, P. Cointe, A generative programming approach to developing dsl compilers, in: R. Glück, M. Lowry (Eds.), 4th International Conference Generative Programming and Component Engineering (GPCE), Springer Berlin Heidelberg, Tallinn, Estonia, 2005, pp. 29–46, doi:10.1007/11561347_4.
[24] S.-B. Cools, C. Gershenson, B. D'Hooghe, Self-organizing traffic lights: A realistic simulation, in: M. Prokopenko (Ed.), Advances in Applied Self-Organizing Systems, Springer, London, 2013, pp. 45–55, doi:10.1007/978-1-4471-5113-5_3.
[25] A. D'Ambrogio, G. Iazeolla, L. Pasini, A. Pieroni, Simulation model building of traffic intersections, Simul. Model. Pract. Theory 17 (2009) 625–640, doi:10.1016/j.simpat.2008.11.001.
[26] J.N. Duarte, J. de Lara, ODiM: A model-driven approach to agent-based simulation., in: Proceedings of the 23rd European Conference on Modelling and Simulation, 2009, pp. 158–165.
[27] S. El-Tantawy, B. Abdulhai, H. Abdelgawad, Design of reinforcement learning parameters for seamless application of adaptive traffic signal control, J. Intell. Transp. Syst. 18 (2014) 227–245, doi:10.1080/15472450.2013.810991.
[28] A. Fernández-Isabel, R. Fuentes-Fernández, Analysis of intelligent transportation systems using model-driven simulations, Sensors 15 (2015) 14116–14141, doi:10.3390/s150614116.
[29] R. Fuentes-Fernández, J.M. Galán, S. Hassan, A. López-Paredes, J. Pavón, 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), Springer, 2010, pp. 81–90, doi:10.1007/978-3-642-12384-9_11.
[30] J.M. Galán, L.R. Izquierdo, S.S. Izquierdo, J.I. Santos, R. del Olmo, A. López-Paredes, B. Edmonds, Errors and artefacts in agent-based modelling, J. Artif. Soc. Simul. 12 (2009) 1.
[31] I. Galorath, SEER by galorath, 2017. http://galorath.com/.
[32] A. Garro, F. Parisi, W. Russo, A process based on the model-driven architecture to enable the definition of platform-independent simulation models, in: N. Pina, J. Kacprzyk, J. Filipe (Eds.), Simulation and Modeling Methodologies, Technologies and Applications, Advances in Intelligent Systems and Computing, volume 197, Springer Berlin Heidelberg, 2013, pp. 113–129, doi:10.1007/978-3-642-34336-0_8.
[33] A. Garro, W. Russo, EasyABMS: a domain-expert oriented methodology for agent-based modeling and simulation, Simul. Modell. Pract. Theory 18 (2010) 1453–1467.
[34] C. Gershenson, Self-organizing traffic lights, Complex Syst. 16 (2005) 29–53.
[35] C. Gershenson, D.A. Rosenblueth, Adaptive self-organization vs static optimization, Kybernetes 41 (2012) 386–403, doi:10.1108/03684921211229479.
[36] C. Gershenson, D.A. Rosenblueth, Self-organizing traffic lights at multiple-street intersections, Complexity 17 (2012) 23–39, doi:10.1002/cplx.20392.
[37] A. Ghorbani, P. Bots, V. Dignum, G. Dijkema, MAIA: a framework for developing agent-based social simulations, J. Artificial Soc. Soc. Simul. 16 (2013) 9.
[38] A. Ghorbani, G.P.J. Dijkema, P. Bots, H. Alderwereld, V. Dignum, Model-driven agent-based simulation: Procedural semantics of a MAIA model, Simul. Modell. Pract. Theory 49 (2014) 27–40, doi:10.1016/j.simpat.2014.07.009.
[39] J.J. Gómez-Sanz, C.R. Fernández, J. Arroyo, Model driven development and simulations with the INGENIAS agent framework, Simul. Modell. Pract. Theory 18 (2010) 1468–1482, doi:10.1016/j.simpat.2010.05.012. http://www.sciencedirect.com/science/article/pii/S1569190X10001024. Simulation-based Design and Evaluation of Multi-Agent Systems.
[40] D. Gračanin, H.L. Singh, S.A. Bohner, M.G. Hinchey, Model-driven architecture for agent-based systems, in: Formal Approaches to Agent-Based Systems: Third International Workshop, FAABS 2004, Greenbelt, MD, April 26–27, 2004, Revised Selected Papers, Springer, Greenbelt, MD, 2004, pp. 249–261. Lecture Notes in Computer Science. 10.1007/978-3-540-30960-4_17.
[41] V. Grimm, U. Berger, D.L. DeAngelis, J.G. Polhill, J. Giske, S.F. Railsback, The odd protocol: a review and first update, Ecol. Model. 221 (2010) 2760–2768.
[42] C. Hahn, A domain specific modeling language for multiagent systems, International Foundation for Autonomous Agents and Multiagent Systems, 2008. Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1, 233–240.
[43] C. Hahn, C. Madrigal-Mora, K. Fischer, A platform-independent metamodel for multiagent systems, Autonomous Agents Multi-Agent Syst. 18 (2009) 239–266, doi:10.1007/s10458-008-9042-0.
[44] L. Hamill, Agent-based modelling: The next 15 years, J. Arti. Soc. Soc. Simul. 13 (2010) 7.
[45] S. Hassan, R. Fuentes-Fernandez, J.M. Galan, A. Lopez-Paredes, J. Pavon, 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), 2009, pp. 1–13.
[46] J. Hutchinson, M. Rouncefield, J. Whittle, Model-driven engineering practices in industry, in: Proceedings of the 33rd International Conference on Software Engineering, ACM, New York, NY, USA, 2011, pp. 633–642, doi:10.1145/1985793.1985882.
[47] T. Iba, Y. Matsuzawa, N. Aoyama, 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, 2004, pp. 1–12.
[48] G.B. Jayatilleke, L. Padgham, M. Winikoff, Evaluating a model driven development toolkit for domain experts to modify agent based systems, in: L. Padgham, F. Zambonelli (Eds.), International Workshop on Agent-Oriented Software Engineering, Springer, Hakodate, Japan, 2006, pp. 190–207. Lecture Notes in Computer Science. 10.1007/978-3-540-70945-9_12.
[49] R. Jensen, An improved macrolevel software development resource estimation model, in: Proceedings of the 5th ISPA Conference, 1983, pp. 88–92.
[50] J. Jin, X. Ma, Adaptive group-based signal control using reinforcement learning with eligibility traces, in: IEEE 18th International Conference on Intelligent Transportation Systems, 2015, pp. 2412–2417, doi:10.1109/ITSC.2015.389.
[51] G. Kahraman, S. Bilgen, A framework for qualitative assessment of domain-specific languages, Softw. Syst. Model. 14 (2015) 1505–1526, doi:10.1007/s10270-013-0387-8.
[52] G. Kardas, Model-driven development of multiagent systems: a survey and evaluation, Knowl. Eng. Rev. 28 (2013) 479–503.

[53] G. Kardas, A. Goknil, O. Dikenelli, N.Y. Topaloglu, Model driven development of semantic web enabled multi-agent systems, Int. J. Cooperative Inf. Syst. 18 (2009) 261–308.
[54] F. Klügl, A.L.C. Bazzan, Agent-based modeling and simulation, AI Mag. 33 (2012) 29–40.
[55] F. Klügl, P. Davidsson, AMASON: Abstract meta-model for agent-based simulation, in: M. Klusch, M. Thimm, M. Paprzycki (Eds.), Multiagent System Technologies, Lecture Notes in Computer Science, volume 8076, Springer Berlin Heidelberg, 2013, pp. 101–114, doi:10.1007/978-3-642-40776-5_11.
[56] T. Kosar, N. Oliveira, M. Mernik, M.J. Pereira, M. Crepinsek, D. Cruz, P. Henriques, Comparing general-purpose and domain-specific languages: An empirical study, ComSIS–Comput. Sci. Inf. Syst. J. (2010) 247–264.
[57] Y. Kubera, P. Mathieu, S. Picault, Interaction-oriented agent simulations: From theory to implementation, in: Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008), IOS Press, Amsterdam, 2008, pp. 383–387.
[58] C. Macal, M. North, Introductory tutorial: Agent-based modeling and simulation, in: Proceedings of the 2014 Winter Simulation Conference, IEEE Press, Piscataway, NJ, USA, 2014, pp. 6–20. In WSC '14
[59] P. Mannion, J. Duggan, E. Howley, An experimental review of reinforcement learning algorithms for adaptive traffic signal control, in: L.T. McCluskey, A. Kotsialos, P.J. Müller, F. Klügl, O. Rana, R. Schumann (Eds.), Autonomic Road Transport Support Systems, Springer, 2016, pp. 47–66, doi:10.1007/978-3-319-25808-9_4.
[60] M. Mernik, J. Heering, A.M. Sloane, When and how to develop domain-specific languages, ACM Comput. Surv. (CSUR) 37 (2005) 316–344.
[61] D. Moreira, F. Santos, M. Barbieri, I. Nunes, A.L.C. Bazzan, ABStractme: Modularized environment modeling in agent-based simulations, in: S. Das, E. Durfee, K. Larson, M. Winikoff (Eds.), Proceedings of the 16th International Conference on Autonomous Agents and Multiagent Systems, IFAAMAS, S ao Paulo, 2017, pp. 1802–1804.
[62] O.M. GROUP, Model driven architecture (MDA): MDA guide rev. 2.0, 2014, http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf.
[63] D.r.d. Oliveira, A.L.C. Bazzan, Multiagent learning on traffic lights control: effects of using shared information, in: A.L.C. Bazzan, F. Klügl (Eds.), Multi--Agent Systems for Traffic and Transportation, IGI Global, Hershey, PA, 2009, pp. 307–321.
[64] J. Osis, E. Asnina, Model-driven domain analysis and software development: Architectures and functions, 1st, IGI Global, Hershey, PA, USA, 2010.
[65] H.V.D. Parunak, R. Savit, R.L. Riolo, Agent-based modeling vs. equation-based modeling: A case study and users' guide, in: J.S. Sichman, R. Conte, N. Gilbert (Eds.), International Workshop on Multi-Agent Systems and Agent-Based Simulation, Springer, Paris, France, 1998, pp. 10–25. Lecture Notes in Computer Science. 10.1007/10692956_2.
[66] J. Pavón, J. Gómez-Sanz, Agent oriented software engineering with INGENIAS, in: V. Mařík, M. Pěchouček, J. Müller (Eds.), International Central and Eastern European Conference on Multi-Agent Systems, Springer, Prague, Czech Republic, 2003, pp. 394–403. Lecture Notes in Computer Science
[67] J. Pavón, J. Gómez-Sanz, R. Fuentes, Model driven development of multi-agent systems, in: A. Rensink, J. Warmer (Eds.), Model Driven Architecture – Foundations and Applications, Lecture Notes in Computer Science, volume 4066, Springer Berlin Heidelberg, 2006, pp. 284–298, doi:10.1007/11787044_22.
[68] L. Penserini, A. Perini, A. Susi, J. Mylopoulos, From stakeholder intentions to software agent implementations, in: E. Dubois, K. Pohl (Eds.), Proceedings of the International Conference on Advanced Information Systems Engineering, Springer, Luxembourg, 2006, pp. 465–479. Lecture Notes in Computer Science. 10.1007/11767138_31.
[69] A. Perini, A. Susi, Automating model transformations in agent-oriented modelling, in: J.P. Müller, F. Zambonelli (Eds.), International Workshop on Agent-Oriented Software Engineering, Springer, Utrecht, The Netherlands, 2005, pp. 167–178. In Lecture Notes in Computer Science. 10.1007/11752660_13.
[70] L.H. Putnam, W. Myers, Measures for excellence: reliable software on time, within budget, 1st, Prentice Hall Professional Technical Reference, 1991.
[71] J. Raphael, E.I. Sklar, S. Maskell, An empirical investigation of adaptive traffic control parameters, in: Proceedings of Ninth International Workshop on Agents in Traffic and Transportation, New York, USA, 2016, pp. 1–8.
[72] P. Ribino, V. Seidita, C. Lodato, S. Lopes, M. Cossentino, Common and domain-specific metamodel elements for problem description in simulation problems, in: 2014 Federated Conference on Computer Science and Information Systems, 2014, pp. 1467–1476, doi:10.15439/2014F298.
[73] R.J. Rossetti, R.H. Bordini, A.L. Bazzan, S. Bampi, R. Liu, D.V. Vliet, Using {BDI} agents to improve driver modelling in a commuter scenario, Transp. Res. Part C 10 (2002) 373–398, doi:10.1016/S0968-090X(02)00027-X.
[74] S. Rougemaille, F. Migeon, C. Maurel, M.-P. Gleizes, Model driven engineering for designing adaptive multi-agents systems, in: A. Artikis, G. O'Hare, K. Stathis, G. Vouros (Eds.), Engineering Societies in the Agents World VIII, Lecture Notes in Computer Science, volume 4995, Springer Berlin Heidelberg, 2008, pp. 318–332, doi:10.1007/978-3-540-87654-0_18.
[75] C. Sansores, J. Pavón, Agent-based simulation replication: A model driven architecture approach, in: Proceedings of the 4th Mexican International Conference on Artificial Intelligence (MICAI), Monterrey, Mexico, 2005, pp. 244–253, doi:10.1007/11579427_25. Lecture Notes in Computer Sience
[76] C. Sansores, J. Pavón, J. Gómez-Sanz, Visual modeling for complex agent-based simulation systems, in: International Workshop on Multi-Agent Systems and Agent-Based Simulation, Springer, 2005, pp. 174–189.
[77] F. Santos, I. Nunes, A. Bazzan, 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, 2017. 170–179.
[78] D. Schmidt, Model-driven engineering, Comput. 39 (2006) 25–31, doi:10.1109/MC.2006.58.
[79] R. Soley, Model Driven Architecture, Object Management Group (OMG) White Paper, 2000. http://www.omg.org/cgi-bin/doc?omg/00-11-05.pdf.
[80] J. Sprinkle, M. Mernik, J.P. Tolvanen, D. Spinellis, Guest editors' introduction: what kinds of nails need a domain-specific hammer? IEEE Softw. 26 (2009) 15–18, doi:10.1109/MS.2009.92.
[81] T. Stahl, M. Völter, J. Bettin, A. Haase, S. Helsen, Model-driven software development: Technology, engineering, management, John Wiley & Sons, 2006.
[82] M. Strembeck, U. Zdun, An approach for the systematic development of domain-specific languages, Softw. 39 (2009) 1253–1292, doi:10.1002/spe.936.
[83] I.J.P.M. Timóteo, M.R. Araújo, R.J.F. Rossetti, E.C. Oliveira, TraSMAPI: An API oriented towards multi-agent systems real-time interaction with multiple traffic simulators, in: International IEEE Conference on Intelligent Transportation Systems, 2010, pp. 1183–1188, doi:10.1109/ITSC.2010.5625238.
[84] J.-P. Tolvanen, S. Kelly, 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), 2016, pp. 711–719.
[85] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L.C. Kats, E. Visser, G. Wachsmuth, DSL engineering: designing, implementing and using domain-specific languages, CreateSpace Independent Publishing Platform, 2013. http://www.dslbook.org
[86] S. Warwas, C. Hahn, The dsml4mas development environment, Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2 (2009) 1379–1380.
[87] S. Warwas, C. Hahn, The platform independent modeling language for multiagent systems, in: K. Fischer, J.P. Müller, J. Odell, A.J. Berre (Eds.), Agent-Based Technologies and Applications for Enterprise Interoperability: International Workshops, ATOP 2005 Utrecht, The Netherlands, July 25–26, 2005, and ATOP 2008, Estoril, Portugal, May 12–13, 2008, Revised Selected Papers, Springer, 2009, pp. 129–153, doi:10.1007/978-3-642-01668-4_8.
[88] C.J.C.H. Watkins, P. Dayan, Q-learning, Mach. Learn. 8 (1992) 279–292.
[89] M.P. Wellman, Putting the agent in agent-based modeling, Auton. Agent Multi–Agent Syst. 30 (2016) 1175–1189, doi:10.1007/s10458-016-9336-6.
[90] M. Wiering, Multi-agent reinforcement learning for traffic light control, in: Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), 2000, pp. 1151–1158.
[91] U. Wilensky, NetLogo, 1999. http://www.ccl.northwestern.edu/netlogo/ center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.
[92] U. Wilensky, Netlogo traffic grid model, 2003, (Center for Connected Learning and Computer-Based Modeling, Northwestern University). http://www.ccl.northwestern.edu/netlogo/models/TrafficGrid.
[93] L. Xiao, D. Greer, Adaptive agent model: software adaptivity using an agent-oriented model-driven architecture, Inf. Softw. Technol. 51 (2009) 109–137, doi:10.1016/j.infsof.2008.02.002. . Special Section - Most Cited Articles in 2002 and Regular Research Papers

[94] I. Zinnikus, G. Benguria, B. Elvesæter, K. Fischer, J. Vayssière, A model driven approach to agent-based service-oriented architectures, in: Proceedings of the German Conference on Multiagent System Technologies, Springer, Erfurt, Germany, 2006, pp. 110–122. In Lecture Notes in Computer Science. 10.1007/11872283_10.