Towards reducing complexity of multi-agent simulations by applying model-driven techniques

Benjamin Hoffmann[1], Kevin Chalmers[2], Neil Urquhart[2], Thomas Farrenkopf[1] and Michael Guckert[1]

[1] KITE - Kompetenzzentrum für Informationstechnologie,
Technische Hochschule Mittelhessen, Germany
{benjamin.hoffmann, thomas.farrenkopf, michael.guckert}@mnd.thm.de
[2] School of Computing, Edinburgh Napier University, Scotland
{k.chalmers, n.urquhart}@napier.ac.uk

Abstract. Creating multi-agent simulations is a challenging task often requiring programming skills at the professional software developer level. Model driven methods of software development are an appropriate tool for reducing the complexity of the development process of such simulations. The modeller is relieved from implementing time consuming programming details and can concentrate on the application itself. We present the domain specific language Athos with which network based traffic simulations can be created declaratively. The models are platform independent and executable code can be generated for two popular multi-agent platforms. We use a simple yet illustrative example to show how Athos can be applied.

Keywords: domain-specific language, model-driven development, traffic simulation

1 Introduction and Motivation

Agent-based simulations are an effective technique to model and analyse systems in which the overall behaviour is determined by the behaviour of its constituent autonomous entities [28].Generally, Agent-Based Modelling (ABM) is a challenging task [25, 31, 5]. In comparison to more traditional modelling approaches like system dynamics, ABM has an increased level of cognitive complexity. Vendrov et al. argue that one reason for the higher complexity can be found in the programming languages used in current ABM frameworks [31]. These languages provide a low-level abstraction which makes it hard to create, understand and validate agent-based models. This is especially true for domain experts who are not professional programmers. Additionally, the complexity of ABM has increased due to grown computing power allowing larger models with more complex interactions [25]. At the same time, the domains analysed by ABM have become more complex themselves and thus require more powerful models [1]. Development of agent-based simulations requires insight into two distinct bodies of knowledge [3]: the knowledge and experience of experts in the respective domain; and skills in software development.

It seems obvious that domain experts should work closely with software engineers to achieve high-quality agent-based simulations. In practice, the cooperation often leads to new problems like inconsistent terminologies [22] or different expectations regarding competences [29]. Therefore, scientific software is often designed and implemented by the same scientists who will eventually use it for their work. These scientists usually lack fundamental software engineering expertise [18]. North and Macal consider the development of interfaces that allow domain experts with little programming knowledge to create agent-based models as one key challenge in the field of ABM [23].

Having successfully implemented an agent-based model, verification and validation pose the next challenges [6]. While verification ensures that a model implementation is congruent with its conceptual specification, validation guarantees that the implementation accurately represents reality [33, 17]. If implementation languages require an increased effort to understand a model implementation, they also make verification and validation more difficult. In addition, current approaches impede replication as they are often directly interwoven with their target platform [28].

The use of Domain-Specific Languages (DSLs) constitutes a promising approach towards the solution of the aforementioned problems (c.f., e.g. [23, 21]). By production of intermediate models that bridge the gap between the researcher's abstract non-formal models and the final model implementation, DSLs present well-sized building blocks to create models from which simulations can automatically be generated. As the complexity of implementation details is hidden from language users, models become more comprehensible and focused. Increased abstraction also facilitates reproducibility of models and thus ensures high-quality simulation results in the scientific community.

In this paper, we present a DSL called **Athos** which supports researchers in the development of agent-based simulations. Athos allows declarative specifications for *traffic simulation* scenarios and thus relieves modellers from complex programming tasks. Major benefits of this approach are a clearer separation of concerns in the overall simulation model and enhanced support for domain experts in the set-up of agent-based simulations.

The remainder of this paper is organized as follows. Section 2 provides background on DSLs and agent-based simulations and discusses how these research fields have been brought together in related projects. Section 3 explains how Athos raises the abstraction level for agent-based models and thus alleviates the aforementioned problems. Section 4 presents a case study that applies Athos to a simple yet illustrative problem. Section 5 concludes this paper and shows some directions for future work.

2 Background and Related Work

A Domain Specific Language (DSL) is a programming language that is tailored towards a specific problem domain. Advantages of DSLs include decreased development time and reduced requirements of software development skills, allowing

a domain expert to develop software for the given domain. When constructing models, a DSL allows a stronger focus on the model and domain issues rather than on lower-level programming issues ([10] and [7]). Exploring complex problems through computer based simulation and modelling has been widely recognised as a useful means of increasing our understanding of real-world problems. A number of software frameworks for simulation and the associated analysis exist. These range from generalist frameworks such as SimStudio [30] to domain specific simulations such as MatSim [16] designed specifically for the transportation domain. A recent overview of the development and use of domain-specific modelling is given by Çetinkaya [4]. According to [16], such software models may be utilised via software packages aiming directly at the domain user. This often leads to models constrained to the functionality offered by the packages' user interface. If a model is incorporated within a DSL, the user may use the DSL to configure the model offering a far wider range of possibilities.

Multi-Agent Systems (MAS) are regarded as a useful means to construct simulations. Most simulations revolve around the interactions of a set of entities such as: people or vehicles [11]; stock market shares [24]); or consumers within a marketplace [9]. In such scenarios, the entities in the system being modelled can be represented by specific software agents. Ge and Polhill [11] use a MAS to investigate the actions of commuters and how their decisions are influenced by changes in the road network (specifically, the addition of new road links). In this case groups of commuters are represented by a software agent that takes decisions from the perspective of that individual.

Most software-based simulations are carried out for the benefit of specialists in other fields. This leads to a paradox. The non-computing specialist is unable to construct simulations using a framework such as the Java Agent Development Environment (JADE) [2] and instead uses an existing simulation package (e.g. MatSim [16]). Package-based simulation is restricted by the functionality provided by the package which may or may not be sufficient for the task being considered. If no package exists that can support the desired simulation, then a platform such as JADE or NetLogo [32] must be used by someone with the appropriate specialised skills. A DSL is a programming language that is tailored towards a specific domain (including Software Engineering [7] or MAS [12]). It requires less skill than a traditional programming language. This gives two distinct advantages: the language can be learned by a domain expert; and the process of model development is simplified.

Within transportation, a DSL has the potential to isolate the modeller from issues such as handling details of maps, coordinate systems or routing. This frees the modeller to concentrate on issues such as defining decision making and making explicit the problem constraints. This has two useful outcomes. Firstly, the development process may be accelerated. Secondly, the development may be undertaken by a domain specialist rather than a software engineer.

A contribution in that direction is made by Hassan et al. [14] who define a process to develop models for a given domain in the field of social sciences. The authors base their approach on the INGENIAS methodology [26] that comprises

a meta-metamodel from which further metamodels can be derived. INGENIAS provides the necessary tools to develop a working graphic editor for the defined (meta-)models together with the necessary transformations to the intended target platform. The presented approach is designed so that it can potentially be applied to any agent-based modelling and simulation approach within the confines of social sciences. On the one hand, this makes the presented approach highly flexible and applicable for a wide range of problems. On the other hand, , it requires users to develop a suitable metamodel for their problem as well as suitable transformations and cannot be applied in an out-of-box style.

GAMA [13] is another contribution that aims to facilitate the creation and simulation of complex agent-based models through employment of a platform-internal DSL. Models are described in GAML, a DSL that allows to define multiple layers within a simulation (e.g. one layer for the inside of buildings and another for the outside world buildings are located in), handling of geographical data, and definition of agents' attributes and behaviour. Though GAML is an agent-oriented DSL, it also allows to describe parts of a model by means of ordinary differential equations. In contrast to our approach, GAML is domain-specific in that it is agent-oriented, but it does not aim at a specific application domain. This makes it difficult to describe models in a pure or mostly declarative way but requires users to define agents' behaviour through *actions* and *reflexes* in a more procedural manner.

3 Athos – A DSL for Traffic Simulations

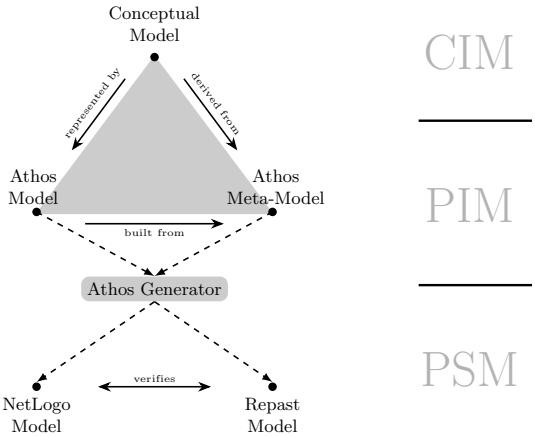

Fig. 1. Athos' Modelling Approach.

Athos is a DSL that seeks to support the development of MAS simulations. The language focuses on traffic scenarios that involve vehicles (agents) with individual behaviour (e.g. finding shortest routes). Thus, an Athos program involves a multitude of individual-level optimisation problems that each affect the state

of the global system. Athos allows scenario definition in a declarative manner. This relieves users from complex programming tasks and enables them to focus on *what* to simulate instead of *how* to simulate it (c.f. [31, 3]).

Figure 1 illustrates the main components of the language and its flow of information. The creation of a traffic simulation with Athos is based upon the development of a conceptual model which features aspects from the domain of traffic optimisation. These models allow to describe information on aspects that are relevant in this context, e.g. the capacity of certain roads in an area of interest or how certain types of vehicles congest certain roads. However, these models do not contain any information on computational details. Thus, from a computational view, they are created on the most abstract level and considered as Computationally Independent Models (CIM) [19].

The language elements of Athos were derived from the CIM level in order to make them available to models which also consider aspects that are relevant for computer simulations. While these models are more specific than their CIM ancestors, they still do not allow any assumptions with regard to the implementation platform. Models on this level are said to be Platform Independent Models (PIM) [19, 28]. Every program written in Athos constitutes such a PIM. Since the meta-model elements from which Athos programs are built are directly derived from the domain of traffic optimisation problems, each Athos PIM concisely represents the underlying CIM of the traffic domain.

The Athos generator finally processes models from the PIM level and transforms them into models for a specific simulation platform known as Platform Specific Models (PSM). In order to transform a PIM into a PSM, the generator has to add platform-specific details to the information drawn from the PIM. This is done by means of code templates which are created for every supported target platform. Currently, Athos features templates for two agent-based simulation platforms. The first is the NetLogo[3] platform, whose models follow a procedural paradigm. The other platform is the Repast Simphony[4] platform for which models are constructed in an object-oriented way. As is pointed out by Sansores and Pavón, the creation of models for different target platforms supports validation and verification efforts as both simulation implementations should present equivalent results [28].

The main tool used in the development of the language is the Xtext[5] language workbench (version 2.12). Next to the definition of the abstract and concrete syntax of the language, the workbench was also used to define transformations that constitute the language's dynamic semantics.

Athos is developed in an iterative and incremental manner. Although Athos is ultimately intended to support the development of various different optimisation-related traffic scenarios, it focuses on one part of the problem domain at a time. The language's first major development iteration focused on scenarios in which agents seek to get from a given starting point in a road network to a predefined

[3] https://ccl.northwestern.edu/netlogo/
[4] https://repast.github.io/download.html
[5] https://www.eclipse.org/Xtext/

destination. For this, they seek to find the route that requires the least amount of time. Since in the underlying network the amount of time it takes to travel a given road is dependent on the current traffic situation, each agent is confronted with a dynamic optimisation problem.

Traffic-related problems necessitate the definition of some kind of network. Athos allows specification of road networks that consist of nodes and edges in a straightforward manner. In order to populate the network with agents, nodes can be defined as sources from which new agents originate. Users can specify time-based patterns in which agents are created. It is also possible to model distribution functions. These functions control how source nodes spawn agents with different properties into the system.

Agents differ in two major properties. Firstly, agents can be assigned different routing modes that determine how they move inside the network. Most of the time, modellers will assign an a priori destination to agents. However, in order to generate background noise or to simulate public transport routes, it is also possible to define agents that circle or shuttle along a pre-defined sequence of nodes in the network. Secondly, agents can be assigned a congestion factor. The congestion factor is a value that determines to what extend the respective agent will congest, i.e. slow down, traffic on the road it travels on. This way, agents can represent different types of vehicles. An agent with a high congestion factor could represent a bus or a tractor whereas an agent with a low congestion factor could represent a fast car or a motorcycle.

As was already stated, agents that head towards a pre-defined destination try to get there in the least possible amount of time. Vehicle agents calculate the fastest path from their current location to their intended destination node by means of Dijkstra's algorithm [8]. They do so every time they enter a node of the network. Whenever agents recalculate the fastest path, they consider the current traffic situation in the network. For this, roads must be assigned a numerical value that represents the amount of time cars need to travel them. To this end, Athos allows the definition of cost functions as ordinary mathematical expressions. Within these expressions, various properties like the length of the road or the accumulated congestion factor of all vehicles can be used. The value of the accumulated congestion factor for a given road depends on both the number and the types of agents that are on this road at a given point in time.

The described language features allow for the definition of traffic-dependent travel durations. The higher the accumulated congestion factors of all vehicles on a given road, the longer it takes these vehicles to get to the next road. This, in turn, increases the time window in which other vehicles can further increase the accumulated congestion factor on this road. This way, the language allows for the definition of scenarios where increased numbers of agents congest certain roads so that traffic may ultimately grind to a halt. By default, all simulation scenarios track the total amount of time cars have spent in the system.

Figure 2 shows an excerpt of the language's meta-model. Any program written in Athos is a `Model`. A program features several types of `Function`s. Some of these are used to influence the way an agent travels a given edge. Such functions

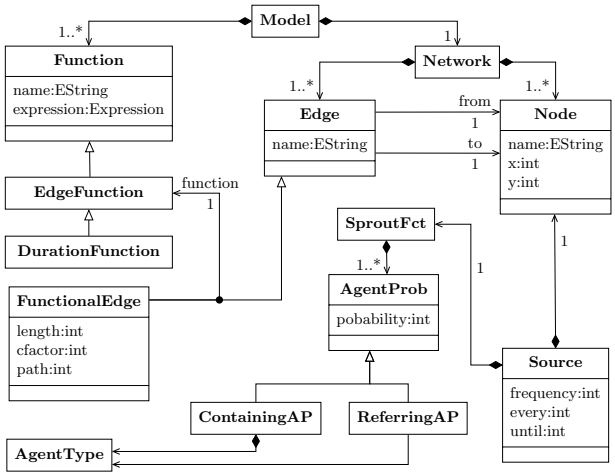

Fig. 2. Simplified Excerpt of Athos' Meta-model

are called `EdgeFunction`s. This is because they are always associated with an arbitrary number of `Edge`s. Agents that travel an edge have to follow the rules implied by the function associated with this edge. As will be shown shortly, there is also another type of function that is used by (or associated with) agents. `EdgeFunction`s are further specialised to `DurationFunction`s. They feature an expression that determines the amount of time it takes to cross a certain edge. Later versions will feature additional specialisations of the `EdgeFunction` class, e.g. speed-dependent functions whose expressions determine the speed by which agents drive on a given edge.

Each program must also feature a `Network` which consists of `Nodes` and `Edges`. Nodes and edges must be identifiable by a unique `name`. Additionally, nodes need coordinates to locate them in the plane. Edges need both a node `from` which they emerge and a node `to` which they lead. Nodes can assume the role of a `Source` that sprouts agents into the network. Source nodes are always associated with a sprout function (`SproutFct`). A sprout function contains a (non-empty) set of agent probabilities (`AgentProb`). Both specialisations, i.e. `ContainingAP` and `ReferringAP`, have a reference to an `AgentType` and an integer that represents the probability with which the referred type of agent is created. While a `ContainingAP` allows to define an anonymous agent type inside a sprout function, `ReferringAP` objects refer to a named agent type defined in an Athos program.

Figure 3 illustrates how Athos allows for the definition of agent types together with type specific attributes and optimisation functions. An `AgentType` can either be named or anonymous depending on whether its `name` attribute was set. Each `AgentType` can be assigned an arbitrary number of attributes defined in the model (e.g. fuelConsumption). The connection between the type and the attribute is established by means of an `AttributeAssignment` which also assigns a value for the attribute. For this reason, all agents of the same type have the

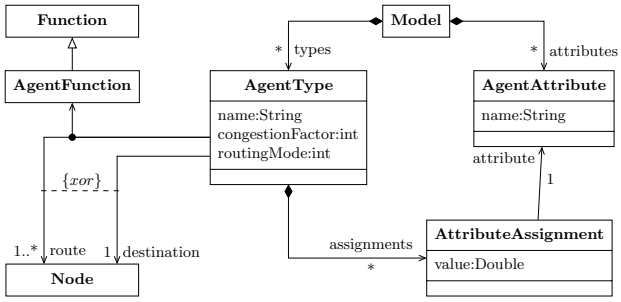

Fig. 3. Athos' `AgentType`-related meta-model elements.

same value for a given attribute. As was already mentioned, different `AgentType`s can be assigned different `congestionFactor`s and `routingMode`s. The attributes of an agent can be used inside an `AgentFunction`. An `AgentFunction` can be assigned to an `AgentType` so that all agents of this type will calculate their routes in a way that minimises the associated function.

4 Example

This case study demonstrates how Athos can be applied to a simple yet illustrative problem. We first consider a network of roads with a single source and a single target of traffic. Agents (i.e. cars) are created in the source node and use a shortest route to the target while travel time depends on congestion effects on the roads represented by the edges of the network.

Given such a traffic aware network with vehicles travelling from defined sources to individually defined targets, analysing the overall behaviour of this system soon becomes a complex task. While static instances (i.e. networks with static travel times not affected by congestion) of the problem can be solved with analytical methods, agent based simulations are an appropriate tool for dynamic versions of this problem. However, even simple networks quickly lead to complex programs in popular, rather easy to use agent based programming environments. In our case study we consider a network of eight nodes and nine edges. This network corresponds to Samuelson's example [27] in which the Braess paradox and the effect of modified cost functions determining the choice of the routes is examined.

```
1  model Model1 world xmax 60 xmin 0 ymax 60 ymin 0
2  functions // functions used in the model
3  durationFunction traveltime  length + cfactor * accCongestionFactor
4  network
5  nodes // nodes of the network
6    node a (2.0, 15.0)
7    node b (10.0, 15.0)
8    node c (20.0, 15.0)
9    node d0 (30.0, 15.0)
10   node t1 (8.0,25.0)
11   node t2 (15.0, 25.0)
12   node d1 (15.0, 5.0)
13   node d2 (23.0, 5.0)
14 edges // roads that connect the nodes
15   edge ab from a to b length 2.0 cfactor 0.02  function traveltime
16   edge cd from c to d0 length 2.0 cfactor 0.02 function traveltime
17   edge at1 from a to t1 length 10.0 cfactor 0.002 path "ac" function traveltime
18   edge t1t2 from t1 to t2 length 10.0 cfactor 0.002 path "ac"  function traveltime
```

```
19    edge t2c from t2 to c length 5.0 cfactor 0.002 path "ac"  function traveltime
20    edge bd1 from b to d1 length 10.0 cfactor 0.002 path "bd"  function traveltime
21    edge d1d2 from d1 to d2 length 10.0 cfactor 0.002 path "bd" function traveltime
22    edge d2d from d2 to d0 length 5.0 cfactor 0.002 path "bd"  function traveltime
23    edge bc from b to c length 1.0 cfactor 0.03 baseSpeed 1.0 function traveltime
24  sources
25     a sprouts (congestionFactor 100.0 destination d0 ) frequency 25.0 every 1 until 3
```

Individual travel time is calculated by means of the duration function *traveltime* defined in line 3. This function accumulates the product of *cfactor* and *congestionFactor* over all individuals using the edge at the very moment and adds it to the length of the edge. Note that *congestionFactor* is an attribute of the travelling agent that determines how strong this individual agent will add to congestion while *cfactor* is an attribute of the edge and defines the (linear) congestion effect on this edge.

This PIM can either be generated to be run as a NetLogo or a Repast Simphony simulation in their respective environment. While the first offers the easy to use NetLogo system, the latter is created to be run in the scalable Java-based Repast Symphony system. Note that this model is also computation independent as we have argued in the previous chapters: we just define the problem without details of how the solution is to be computed leaving that as a task of the language and the architecture.

Comparing the model above with directly implemented simulations in both of the target platforms shows that the model is highly self-documenting. The code generated for either of the two platforms is difficult to understand even though the underlying templates follow best practice approaches for the platforms. Probably one of the most basic measures for software complexity is simply counting the number of statements or just lines of code. While the Athos model consists of 25 lines, the generated NetLogo source contains 572 lines and the Repast Symphony program 1531 lines in 8 classes. Note that according to the principles of model driven software development, the generated code is well structured and follows best practice guidelines and does not contain unnecessary statements.

We will discuss some exemplary language features of Athos by extending the model from above. For example, different types of agents with individual behaviour in how they determine their preferred route from source to destination can be defined. This can be done very intuitively by just defining an additional type of agent *ecoDriver* to the default type and an alternative function *ecoDriverFunction* to compute travel costs relevant to this type of agent. *ecoDriver* agents possess the attribute *fuelConsumption* available to the newly defined function.

```
1  ...
2  agentAttributes fuelConsumption
3  agentTypes
4  agentType ecoDriver congestionFactor 50.0 attr fuelConsumption = 10.0 destination d_ optimises ecoDriverFunction
5  functions // functions used in the model
6  durationFunction traveltime  length + cfactor * accCongestionFactor
7  agentFunction ecoDriverFunction length * fuelConsumption
8  ...
```

Furthermore, we can define two sources of traffic (a and b) both being the origin of agents heading towards the same destination d. This is achieved by replacing the last line of the model code by:

```
1  b sprouts (congestionFactor 100.0 destination d0 ) frequency 50.0 every 1 until 1
2  a sprouts (ecoDriver probability 95) , (congestionFactor 100.0 destination d0 probability 5) frequency 50.0 every 1 until 1
```

This increases the number of lines of code in the Athos model to 30 and to 587 in the generated NetLogo source.

Besides the number of lines of code as an obvious indicator for complexity, a closer inspection of the implementations shows how the code in both platforms contains a significant amount of instructions that can hardly be mapped to the simulation directly but are necessary implementation details a domain expert would not want to bother with. Following Crooks et. al., by applying the rule of parsimony (aka Occam's razor) we again get a strong argument for the quality of the Athos approach (see [6]).

5 Conclusion and Future Work

This paper argues that a model driven approach for MAS in a given domain (e.g. traffic as in our case study) can significantly reduce the complexity of the code that is visible to the domain expert. The domain expert creates platform independent models that are transformed into executable simulations. In this paper, we have shown how Athos allows the creation of PIMs from computationally independent conceptual models in order to define scenarios in which agents solve optimisation problems. In order to provide a deeper understanding of the language, we have discussed Athos' meta-model elements and how they are related. We have also shown that Athos models describe the underlying problem in a concise and declarative way. While the optimisation problems solved by the agents are rather simple on the individual level, they yield a complex system behaviour as all agents mutually affect their travel times in the network.

Athos was developed with an iterative approach and will continuously be extended in that way. Later versions of the language will allow to define agents that receive information on the current state of the network in a temporally deferred manner. This will allow experiments on the importance of timeliness of information in traffic networks. We will also introduce features that give fine-granular control on what variables are to be tracked and visualised. Currently, we are working on agents that can find optimal tours for a given set of nodes and congestion dependent travel times in the network, i.e. these agents are facing instantiations of a dynamic Travelling Sales Problem.

Even though the number of lines of code required to define optimisation-related traffic simulations can be dramatically reduced, there is a need to quantitatively evaluate how this affects the cognitive complexity of the models. For this, Athos will have to be evaluated in an objective way by appropriate tests. In this context, we will also empirically evaluate qualitative DSL aspects like usability, productivity, learnability and reliability that are crucial for a successful DSL [15]. We consider such a quantitative evaluation of these aspects to be of utmost importance even though it is often neglected by language developers [20].

However, literature still offers little guidance on how how DSLs can be systematically evaluated [20]. This might be because this part of DSL development is still at a rather immature stage [5]. For this reason, scientists should view it as an important and interesting and rather unexplored area of further research.

References

1. Ana L. C. Bazzan and Franziska Klügl. A review on agent-based technology for traffic and transportation. *The Knowledge Engineering Review*, 29(03):375–403, 2014.
2. Fabio Bellifemine, Federico Bergenti, Giovanni Caire, and Agostino Poggi. *Jade — A Java Agent Development Framework*, pages 125–147. Springer US, Boston, MA, 2005.
3. David Bruce Borenstein. Nanoverse: A constraints-based declarative framework for rapid agent-based modeling. In Levent Yilmaz, editor, *Proceedings of the 2015 Winter Simulation Conference*, pages 206–217, Piscataway, NJ, 2015. IEEE.
4. D. Çetinkaya. A model driven approach to web-based traffic simulation. In *A model driven approach to Web-based traffic simulation*, 2016.
5. Moharram Challenger, Geylani Kardas, and Bedir Tekinerdogan. A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems. *Software Quality Journal*, pages 1–41, 2015.
6. Andrew Crooks, Christian Castle, and Michael Batty. Key challenges in agent-based modelling for geo-spatial simulation. *Computers, Environment and Urban Systems*, 32(6):417–430, 2008.
7. Arie Van Deursen and Paul Klint. Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998.
8. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
9. Thomas Farrenkopf, Michael Guckert, Neil Urquhart, and Simon Wells. Ontology based business simulations. *Journal of Artificial Societies and Social Simulation*, 19(4):14, 2016.
10. Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.
11. Jiaqi Ge and Gary Polhill. Exploring the combined effect of factors influencing commuting patterns and co2 emissions in aberdeen using an agent-based model. *Journal of Artificial Societies and Social Simulation*, 19(3), jun 2016.
12. R Grey. Agent tcl: A transportable agent system. *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information andKnowledge Management (CIKM95)*, 1995.
13. Arnaud Grignard, Patrick Taillandier, Benoit Gaudou, Duc An Vo, Nghi Quang Huynh, and Alexis Drogoul. Gama 1.6: Advancing the art of complex agent-based modeling and simulation. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 117–131, 2013.
14. Samer Hassan, Rubén Fuentes-Fernández, José M. Galán, Adolfo López-Paredes, and Juan Pavón. Reducing the modeling gap: On the use of metamodels in agent-based simulation. In *6th conference of the european social simulation association (ESSA 2009)*, pages 1–13, 2009.
15. Felienne Hermans, Martin Pinzger, and Arie van Deursen. Domain-specific languages in practice: A user study on the success factors. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems: 12th International Conference*, pages 423–437. Springer Berlin Heidelberg, 2009.
16. A. Horni, K. Nagel, and K.W. Axhausen. *The Multi-Agent Transport Simulation MATSim*. London: Ubiquity Press., 2016. DOI: http://dx.doi.org/10.5334/baw.
17. Michael J. North and C. Macal. Agents up close. In Michael John North and Charles M. Macal, editors, *Managing business complexity*, pages 24–44. Oxford University Press, Oxford and New York, 2007.

18. Lucas N. Joppa, Greg McInerny, Richard Harper, Lara Salido, Kenji Takeda, Kenton O'hara, David Gavaghan, and Stephen Emmott. Troubling trends in scientific software use. *Science*, 340(6134):814–815, 2013.
19. Martin Kardoš and Matilda Drozdová. Analytical method of cim to pim transformation in model driven architecture (mda). *Journal of information and organizational sciences*, 34(1):89–99, 2010.
20. Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:77–91, 2016.
21. Tomaž Kosar, Marjan Mernik, and Jeffrey C. Carver. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical software engineering*, 17(3):276–304, 2012.
22. Ruqian Lu and Zhi Jin. Domain modeling-based software engineering: a formal approach. volume 8, page 123. Springer Science & Business Media, 2000.
23. Michael J. North and Charles M. Macal. Agent based modeling and computer languages. In Robert A. Meyers, editor, *Encyclopedia of Complexity and Systems Science*, pages 131–148. Springer New York, New York, NY, 2009.
24. R. G. Palmer, W. Brian Arthur, John H. Holland, and Blake LeBaron. An artificial stock market. *Artificial Life and Robotics*, 3(1):27–31, Mar 1999.
25. Hazel R. Parry. Agent based modeling, large scale simulations. In Robert A. Meyers, editor, *Encyclopedia of Complexity and Systems Science*, pages 148–160. Springer New York, New York, NY, 2009.
26. Juan Pavon, Jorge J. Gomez-Sanz, and Rubén Fuentes. The ingenias methodology and tools. In Brian Henderson-Sellers and Paolo Giorgini, editors, *Agent-Oriented Methodologies*, pages 236–276. IGI Global, 2005.
27. P.A. Samuelson. Tragedy of the open road: Avoiding paradox by use of regulated public utilities that charge corrected knightian tolls. *Journal of International an Comparative Economics*, 1(1):3–12, 1992.
28. Candelaria Sansores and Juan Pavón. Agent-based simulation replication: A model driven architecture approach: Micai 2005: Advances in artificial intelligence: 4th mexican international conference on artificial intelligence. pages 244–253. Springer Berlin Heidelberg, 2005.
29. Judith Segal. Software development cultures and cooperation problems: A field study of the early stages of development of software for a scientific community. *Computer Supported Cooperative Work (CSCW)*, 18(5):581, 2009.
30. Luc Touraille, Mamadou K. Traoré, and David R. C. Hill. A model-driven software environment for modeling, simulation and analysis of complex systems. In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, TMS-DEVS '11, pages 229–237, San Diego, CA, USA, 2011. Society for Computer Simulation International.
31. Ivan Vendrov, Christopher Dutchyn, and Nathaniel D. Osgood. Frabjous: A declarative domain-specific language for agent-based modeling. In William G. Kennedy, Nitin Agarwal, and Shanchieh Jay Yang, editors, *Social computing, behavioral-cultural modeling, and prediction*, volume 8393 of *Lecture notes in computer science*, pages 385–392. Springer, Berlin, 2014.
32. Uri Wilensky. Netlogo. http://ccl.northwestern.edu/netlogo/, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, 1999.
33. Xiaorong Xiang, Ryan Kennedy, Gregory Madey, and Steve Cabaniss. Verification and validation of agent-based scientific simulation models. In *Agent-Directed Simulation Conference*, pages 47–55, 2005.