GeoInformatica DOI 10.1007/s10707-015-0233-y

A Domain Specific Language for Spatial Simulation Scenarios

Luís Moreira de Sousa · Alberto Rodrigues da Silva

Received: 21 December 2013 / Revised version: 19 December 2014 / Accepted: August 2015

Abstract This article describes DSL3S, a domain specific modelling language for Spatial Simulation in the field of Geographic Information Systems (GIS). Techniques such as cellular automata and agent-based modelling have long been used to capture and simulate the temporal dynamics of spatial information. Tools commonly employed to implement spatial simulation models include code libraries and pre-compiled models; the former require advanced programming skills while the latter impose relevant constraints on application scope. Previous attempts to produce domain specific languages in the field have invariably resulted in new textual programming languages (e.g. SELES, NetLogo, Ocelet) that are platform specific and in some cases with weak GIS support and interoperability. DSL3S synthesises relevant concepts of spatial simulation in a UML profile, that allows the design of simulation models through the arrangement of graphical elements. An implementation of this language is also presented, that relies on Model Driven Development (MDD) tools distributed with the Eclipse IDE. This includes a code generation infrastructure, that produces ready to run simulations from DSL3S models, supported by the MASON simulation tool-kit. Finally, DSL3S models for three simple and classical simulations allows to better illustrate and discuss the usage of the language.

Keywords Spatial Simulation · Domain Specific Language · UML Profile · Model-Driven Development

L. M. de Sousa
Luxembourg Institute of Science and Technology
41, rue du Brill
L-4422 Belvaux
Luxembourg
Tel.: +352 42 59 91 3356
Fax: +352 42 59 91 555
E-mail: luis.a.de.sousa@gmail.com

A. R. da Silva
INESC-ID, Instituto Superior Tcnico, Universidade de Lisboa
Rua Alves Redol, 9
1000-029 Lisboa
Portugal
E-mail: alberto.silva@tecnico.ulisboa.pt

1 Introduction

The data stored in an information system usually portraits the world as it was at a specific moment or interval in time. This is especially true for spatial data, but with the added certainty that it will also evolve with time. The patterns of land use and land cover, of social, economic, and demographic variables in general, change constantly with time. Entire organisations have developed with the sole purpose of collecting and updating spatial data, through several data acquisition techniques [Kraak and Ormeling, 2009]. Nevertheless, regular data collection provides at best a periodic picture of the changing reality, which in some applications may not be enough [Batty, 2007]. Stakeholders of an information system may need not only to know how the data changed in the past; in order to plan ahead or otherwise reason upon the data, they also need to understand why it changed the way it did and how it may continue to evolve in the future.

This need is met by specific tools and methodologies composing a niche of spatial analysis called Spatial Simulation [de Smith et al., 2015]. Such tools allow the simulation of space-time changes of spatially distributed variables, usually on a discrete representation of space. Their use can improve knowledge on geographic phenomena in two ways [Batty, 2007]: (1) disclose the dynamics behind changes observed in the past and (2) forecast future evolution and change. The

dynamics identified is typically applied to a set of data during a time period of interest [Law, 2007].

Every since the inception of the first spatial simulation code library in the 1990s, the field experienced rapid growth with an increasing number of tools, such as Swarm, RePast or StarLogo, today comprising perhaps more than one hundred [de Smith et al., 2015]. However, the spatial analyst is thus faced with a non trivial choice of tools, often requiring solid programming skills, at times facing data interoperability issues.

This article presents a Domain Specific Language (DSL) for spatial simulation in the context of GIS, named "*Domain Specific Language for Spatial Simulation Scenarios*" (DSL3S). Additionally, an accompanying development framework is introduced, that allows the development of models through the arrangement of graphical elements and their relationships, dispensing formal programming knowledge. These graphical models can then be translated into ready to run simulations through the application of code generation techniques [Selic, 2003]. The Information Systems Group of INESC-ID[1] research centre associated with the Instituto Superior Técnico of the Universidade de Lisboa, has now close to a decade of experience in this field, applying Model-Driven Development (MDD) and Language Engineering techniques in difference contexts, particularly through the ProjectIT initiative [Silva et al., 2007a,b; Saraiva and Silva, 2009; Ferreira and Silva, 2012; Ribeiro and Silva, 2014].

Section 2 of this article reflects on the need for DSLs in Spatial Simulation; Section 3 reviews and compares previous DSLs in the field. Section 4 describes the development approach and Section 5 outlines the syntax and semantics of the language. Section 6 details the prototype implementation of this language and the technologies supporting it. Section 7 presents three application examples of DSL3S. Finally, Section 8 summarises the article and discusses future work.

2 Background on Spatial Simulation

Cellular Automata is the oldest technique used in Spatial Simulation [Wuensche and Lesser, 1992] in which the world is discretised in a grid of regular cells evolving in accordance to a fixed set of rules. More recently, Agent-based Modelling has become a popular paradigm, with wide application in the GIS context [Batty, 2007]. An agent can be defined as an autonomous object that perceives and reacts to its environment [Ferber, 1999] [Weiss, 1999], a concept that stems from object oriented programming. Agent-based Modelling and Cellular Automata are two techniques that superimpose to some extent in the GIS context, though the former brought new processing possibilities, with spatial elements not only reacting to stimuli but also storing knowledge and reasoning before acting. Agents can also be used to model phenomena that do not have direct geographic meaning, such as social or economic interactions.

With at least two decades of history in the GIS field, Spatial Simulation has been in good measured barred from regular spatial analysts. To choose an appropriate tool, the analyst faces in first place the option between two categories: those tools providing support at *Program-level* – closer to a programming language approach – and those that operate at *Model-level* – closer to the conceptual level. The first commonly comprise source code libraries usable with high level programming languages. The second consists in pre-programmed tools that only allow users the parametrisation of the model. These two categories can be seen as the ends of a spectrum of support level defined by the multiple tools available in this field. [Fall and Fall, 2001]. At the middle of this spectrum are found DSLs that try to find a balance between the levels of support.

Program-level tools, such as Swarm [Iba, 2013], MASON [Luke et al., 2005] or REPast [North et al., 2005], can reduce some of the burdening of directly using a general purpose programming language, but still require good programming skills from the analyst [Tobias and Hofmann, 2004]. Most of these code libraries are based on object oriented languages, such as Objective-C or Java, by themselves not easily accessible to entry level programmers. The full knowledge of one of these code libraries is something achievable only with several months of practice [Samuelson and Macal, 2006]. There is thus a relevant time lag between the option for one of these tools and a first simulation prototype, which for some projects may not be acceptable.

On the other hand, *Model-level* support tools (e.g. SLEUTH [Clarke et al., 1997], TELSA [Merzenich and Frid, 2005], LANDIS [Mladenoff, 2004]) tend to be quite specific, much of the model behaviour and assumptions are hidden in the software and may not be explicit or modifiable; their use in other application fields is largely impossible. The analyst can in fact dispense programming skills using this kind of tools, but is constrained to a specific field (e.g. Hydrography, Forest Management) and overall simulation behaviour. These tools also tend to narrow the interaction with geo-referenced data, by imposing certain formats or in some cases by lacking output functionality. Moreover, *Model-level* tools tend to impose dependencies on third party software that may not be trivial to overcome. Evolution or gener-

[1] http://www.inesc-id.pt/

alisation of these tools can sometimes become too expensive and fate them to extinction. Traditionally, they take advantage of market niches providing the needs of a specific and restricted group of users, thus the commercial nature of many of them.

In this context, three essential problems and difficulties arise as the motivation for the DSLs in this field:

1. most spatial simulation tools require specialised programming training;
2. those tools that do not require such knowledge are narrow scoped and tend to compromise GIS interoperability; and
3. an integrated approach to the description, documentation and communication of agent-based models is largely lacking [Müller et al., 2014].

Analysts working with spatial data either come from GIS related areas, like Geography, Cartography or Geodesy, or from the scientific domains of application, such as Biology, Economics or Environmental Science. Even higher education programmes on these fields largely lack programming training, particularly on object oriented development. GIS analysts thus generally lack the knowledge and practice of trained programmers, being unable to use the most common Spatial Simulation tools. The involvement of programmers in Spatial Simulation projects becomes indispensable, creating a further communication step between a model concept and its implementation.

On the other hand, the option for pre-compiled *Model-level* tools also imposes its dose of burdens. First of all the correct implementation of this sort of models is often hard or impossible to verify, since most are commercial, or otherwise closed source tools. Experiments with different behaviours or the input of alternative spatial information is impossible, which sometimes leads analysis to conform to the model, where the opposite would be the desired approach.

Lastly, regarding model descriptions, if a model can only be described by the source code that implements it, then it becomes unreadable to most GIS analysts, as per the above. Beyond that, source code specificities, such as data input/output, syntactic structure and programming paradigm, cast a layer of obfuscation that makes it hard to compare different models. There are numerous concepts common to any spatial simulation, such as the succession of time, spatial variables, agents, behaviours or spatial location. For example, a wildfire model can appear entirely different from a land use model simply because different tools were used to implement each concept, even though the basic programming constructs that compose each of them can be the same. Without some sort of common descriptive lexicon, models are harder to compare and communicate, even those produced for the same application domain.

Potential exists for a wider adoption of Spatial Simulation techniques, provided tools that make model development more accessible to non-programmers, together with a common lexicon for their description.

3 Related work

There have been several attempts to create DSLs for Spatial Simulation, trying to bridge the gap between pure code libraries and *Model-level* tools. In this section some of these DSLs are briefly described; a wider review of spatial simulation tools can be found in de Sousa and Silva [2011].

StarLogo started as a specialisation of the Logo functional programming language, directed at Agent-based simulations. It was an educational project at the MIT to help students exploring emergent behaviour. StarLogo was progressively transformed into a multi-platform tool with the adoption of Java as execution environment; eventually it evolved into a spin-off named **NetLogo**. The lexicon of NetLogo is composed of four main concepts, all different kinds of agents: (i) *turtle* - agent capable of moving across the simulation space; (ii) *patch* - a static subdivision of the simulation space; (iii) *link* - a relation between two turtles; (iv) *observer* - a non-spatial agent capable of collecting data from, and provide data to, other agents. Agents can themselves contain *variables* to store data and can be grouped in *agentsets*. A vast library of over 300 pre-built models has been gathered for education purposes[2], covering a wide range of disciplines. Both StarLogo and NetLogo are relatively easy to learn, especially when compared to *Program-level* tools, dispensing the higher skills needed to use an object oriented language [Railsback et al., 2006]. An integrated text editor supports swift development and the exploration of model dynamics. More recently an extension[3] for spatial data input was made available, although entirely reliant on ESRI data formats. Berryman [2008] reports that this extension requires advanced programming skills to master. Of the various DSL attempted in this field, NetLogo seems to be the most popular, retaining a large number of users. In great measure this is due to its fast prototyping capabilities, to which the integrated text editor greatly contributes. However, readability issues common to traditional programming languages slowly emerge with larger and more complex models, especially if spatial data is involved.

[2] http://ccl.northwestern.edu/netlogo
[3] http://ccl.northwestern.edu/netlogo/docs/gis.html

The Spatially Explicit Landscape Event Simulator (**SELES**) is the product of a research project at the Simon Fraser University, a declarative DSL for Landscape Dynamics [Fall and Fall, 2001]. SELES was conceived to be used closely with GIS software, supporting a vast range of different raster formats (most common in Land Use / Land Cover data) for *landscape* data input. SELES takes also as input a set of *global variables* and the declaration of several landscape *events* and *agents*. Landscape events describe the model dynamics, each requiring the declaration of a spatial domain and recurrence frequency. For each event a spreading mechanism is specified and how it affects its neighbourhood. Even though using keywords closer to the context of simulation, simulations coded with SELES are somewhat reminiscent of third generation languages, with distinct data and procedure environments, still leaving many usual coding activities to the user. It is a good example of a DSL that while dealing way with some of the complexity of traditional programming languages, achieves little in terms of abstraction. SELES is shipped with a dedicated code editor and a simulator that runs the model by interpreting the code files and reading in the spatial data. At run time the simulator displays the model in a graphical interface. Both these programs are available free of charge as closed executables for Microsoft operating systems.

MOBIDYC (Modelling Based on Individuals for the Dynamics of Communities) is an Agent-based approach to the study of population dynamics, directed at the fields of Biology and Ecology [Ginot et al., 2002]. It was conceived to provide a tool accessible to non-programmers, particularly biologists. In essence, MOBIDYC is a Smalltalk code package, defining a set of simple primitives, such as *environment*, *agent* and *state*, plus a set of pre-defined *behaviours*. A model requires in first place the creation of agents and their respective states; behaviours are coded with primitive relations between the names of state variables, such as arithmetic operations. Observing agents to collect data can be added, but results are made available only in tabular format. Models developed with MOBiDYC can be quite fluid and easy to understand, if targeting Biology related problems; in other domains the semantics of the code can become harder to grasp. There is no explicit mechanism to interact with GIS software, MOBIDYC was conceived to run primarily on purely artificial spaces. The source code is open and free, but is dependent on VisualWorks, a commercial IDE. The reliance on this IDE provides wide portability to MOBIDYC, running on Microsoft, Macintosh and Linux operating systems.

Ocelet is a declarative DSL for landscape dynamics aimed at tackling common difficulties in capturing space-time dynamics with traditional modelling techniques [Degenne et al., 2009]. It takes an unconventional approach to this field by mimicking the concept of service-oriented architecture, with models composed by components interacting with each other through services. To declare a model with Ocelet, the developer disposes of five principal constructs: (i) *entity* - a component that provides a set of services; (ii) *service* - communication port of an entity, accepting a set of arguments and returning a set of results; (iii) *relation* - bonding entities through their services (when compatible); (iv) *scenario* - describing which relations within an entity have to be activated, and when; (v) *datafacer* - a device through which entities access data. Entity behaviour is coded as actions behind each service through mathematical expressions. The double paradigm of this language presents a novel approach to spatial simulation, but it is not entirely clear if it eases model understanding. Users lacking a background on computer science may find the service-oriented architecture alien and hard to frame with spatial simulation. On the other hand the service-oriented paradigm provides a level of abstraction over the general purpose of a model that is lacking in the other languages reviewed here. However, as the amount of code required to describe larger models expands, this abstraction slowly dilutes. The language is supported by two Eclipse plug-ins: a language editor and a code generator. The artefacts generated are Java classes that can be compiled to specific operating systems or platforms.

In recent years a consortium of French and Vietnamese research centres and universities has developed an agent based modelling IDE called GAMA Grignard et al. [2013]. It is conceived to support large models and to provide seamless integration with spatial data. This IDE interprets a textual DSL called **GAML** (GAMA Modelling Language). A model in GAML is declared in similar fashion to SELES: a structured file composed by sequences of statements that can either be declarative of imperative. The core concept of this language is *Species*, essentially an agent class, that underpins most other concepts. A GAML model is structured into four code sections: (i) *Header* - setting the model name and optionally import other model files; (ii) *Global species* - declaring a special species called "world agent" enclosing global properties of the model; (iii) *Species and grids* - where are declared classes of agents and grid topologies (discrete spatial variables); (iv) *Experiments* - special agents that carry out the execution of the model, being two types: *gui* and *batch*. A species is defined as a set of attributes plus a set of *actions* and *behaviours*.

Behaviours include: *reflex* - a sequence of statements that can be executed at each time step; *init* - a special form of reflex that is evaluated only once when the agent is created; *task* - a reflex with a weight associated that determines its execution priority in the scheduler; and *state* - determines if the agent should enter/leave a particular state at each time step. Beyond four primitive data types (*bool*, *float*, *int* and *string*), GAML supports several advanced features found in general purpose programming languages: loops, iterators and data structures such as lists, maps or matrices. Of the DSL reviewed here GAML is possibly the most versatile, with a wider range of application, due to an extensive number of features and constructs. Eventually, it may come to build a relevant user community like NetLogo did. Nevertheless, as with SELES, GAML still mimics in various ways early third generation languages (such as COBOL) with strict environments for specific code sections. Mastering a language of this depth is naturally a lengthy process, presenting a relevant challenge for less experienced users. GAMA is built on Eclipse, runs on Java and is released under an open source license.

A graphical DSL not conceived for spatial simulation, but worth of mention, is the Agent Modelling Language (**AML**) [Trencansky and Cervenka, 2005]. It was developed for social dynamics and is reliant on the Model Driven Architecture (MDA) infrastructure, extending a wide range of different UML meta-classes. Its concepts are organised hierarchically, through several levels of generalisation. At the top is the concept of *semi-entity*, an abstract element that can be of two types: *behavioured* or *socialised*; the former represents elements that can act on their environment, the later specifies elements that can form societies and participate in social relationships. The concrete building blocks of AML are *entities*, that can be of three types: (i) *agents* - capable of interactions, observations and autonomous behaviour; (ii) *resources* - physical or informational entities whose availability is constrained; and (ii) *environments* - logical or physical surroundings that determine under which conditions entities can exist and function. Three other main concepts model social dynamics: (i) *structures* - to identify societies and roles; (ii) *behaviour* - constructs for communication, observation, reaction and services; and (iii) *attitudes* - to describe individual agent drivers: needs, intentions, goals, beliefs. There is much more to AML, constructs to specify mental agent aspects and even concepts to describe model deployment and execution. No interpreter or code generation infrastructure has ever been developed for AML and no applications could be found in the literature. It is possible that such a detailed language presents too much of a challenge for a full implementation. On the other hand, at the time AML was published, MDA tools where few and less mature than today. AML presents itself as a resource with great potential that is yet to be fulfilled.

Existing DSLs for Spatial Simulation can ease model development and reduce the build-up time in prototyping, but do not fully avoid the need of programming skills. As with general purpose programming languages, the user has to understand the meaning of keywords and how to compose a coherent set of instructions or declarations into a specific model. Some of these DSLs were clearly developed for educational purposes, more as prototyping than analysis tools. Lack of GIS interoperability is an issue to some of them, as so platform or operating system dependency. Apart from AML, these previous DSLs focus on providing a refined concrete syntax but still framed in older programming paradigms emanating from declarative of functional languages.

4 Proposed Approach

The vision of this proposal is to provide GIS analysts means of prototyping spatial simulation models with graphical diagrams, that can be parametrised and tuned to the specific application domain. These graphical models are then feed to a code generation facility to produce a ready-to-run simulation based on one of the popular *Program-level* tools for basic validation. From there analysts can tune the model at the conceptual level using graphical constructs in an iterative process. In this fashion GIS analysts focus their work on modelling itself, abstract of concerns specific to programming, data input or platform dependencies.

Modelling plays an indispensable role in classical engineering disciplines, allowing engineers to study large and complex systems from a higher level of abstraction [Atkinson and Kühne, 2003]. In the still infant field of software engineering, modelling is yet to be widely adopted [Clark and Muller, 2012], although in many cases end users are requiring systems with a degree of complexity that goes well beyond the abilities of traditional software development tools [France and Rumpe, 2007]. Moreover, the integration with parallel disciplines: systems engineering, software engineering, control engineering, business process engineering, etc, can be greatly simplified with proper modelling tools [Giese and Henkler, 2006].

Model-Driven Development[4] (MDD) is a generic designation for several tools and methodologies used to

[4] Also referred in the literature as Model-Driven Engineering

thoroughly include modelling in software engineering [Atkinson and Kühne, 2003]. The successful application of MDD requires a fundamental shift in the way software engineers use models, evolving from *ad hoc* complementary documents to the main focus of their work, thus relegating coding to the background. This is achie-ved through model-to-model and model-to-code transformations, and in some cases by direct model execution. With MDD source code becomes a sub-product of the development process, where the focus is on what the system must do, instead of how it does it [Selic, 2003].

The motivation behind MDD in the software development field is the gain of productivity and quality it can yield through automatic code generation. But further advantages have been identified that justify its application to other domains. In first place the increase in *understandability*, especially since MDD mostly relies on graphical constructs, more expressive by nature, but also for dispensing the text parsing needed to comprehend source code [Selic, 2003]. Secondly, it promotes *fast prototyping*, by allowing model execution from a high level of abstraction, before much effort or resources have to be spent on development. This allows early model validation and later on, during the model refinement process, also to identify unintended or undesired model changes [Selic, 2008; Mohagheghi et al., 2013]. MDD further makes possible the creation of *user-definable mappings*, the capturing of domain specific concepts at an ontological (or meta-model) level, producing a lexicon of model constructs totally independent of particular code languages or specific software platforms [Atkinson and Kühne, 2003; France and Rumpe, 2007]. Finally, it is important to note that a successful MDD application also brings forward an increase in *interoperability*, by offloading such technical concerns on the code generation infrastructure, that can be adapted to match particular environments or platforms [Atkinson and Kühne, 2003; France and Rumpe, 2007].

DSL3S is an application of the MDD philosophy to the specific field of Spatial Simulation, as an alternative way to address the problems identified in Section 2. By raising the level of abstraction at which development takes place, this approach can facilitate the communication between programmers and analysts and other stakeholders lacking programming knowledge [Mohagheghi et al., 2013]. It can also allow prototyping by non-programmers. By detaching model development from specific technologies, it can improve interoperability with geo-spatial data, generating the appropriate code as needed. Lastly, it can lay the foundations for a standard language in the field, as successful efforts in parallel fields have proved, like SysML[5] or ModelicaML[6].

This work employs the Model-Driven Architecture (MDA)[7] methodology, the concrete MDD approach specified by the Object Management Group (OMG). The UML 2.0 modelling language allows the extension of its core primitives (graphical elements, links, etc) through specialisation for different application domains [OMG, 2005]. This is achieved with the definition of a UML Profile, a collection of *stereotypes*, *properties* and *constraints*. Stereotypes are specialisations of existing UML model elements, defining new elements representing narrower abstractions. A semantically related set of stereotypes, specified by properties and restrictions, can thus be used to customise UML into a new specialised language dedicated to a certain domain.

DSL3S takes spatial simulation as a branch of the wider Spatial Analysis GIS field, where model inputs primarily originate from a GIS and whose outputs also have geo-referenced relevance. At this time the language does not contemplate agents with the internal cognitive capacities that Franklin and Grasser [1997] classify as adaptive agents, nor are any explicit concepts of society, or societal interaction considered. All agents are assumed to exist in the space of simulation, thus forcefully being geographic entities. The language does not employ a distinction between Agent-based models and Cellular Automata, aiming at a single approach to both schools of Spatial Simulation, hiding such implementation details from the user.

5 The DSL3S language

DSL3S is defined as a UML profile that includes a set of stereotypes enclosing abstractions underpinning Spatial Simulation. These stereotypes can be seen as the conceptual terms used when explaining a simulation with the terminology of this application domain, e.g., describing “fire” as an agent (because it is mobile and transforms the landscape) or “height” as a spatial variable (that has no innate activity but may influence the actions of certain agents). The DSL3S UML profile allows the development of simulation models by applying these stereotypes and creating the correct relations between them.

This section details the DSL3S language; Section 5.1 presents its Abstract Syntax, Section 5.2 its Concrete Syntax, Section 5.3 lays out the structural semantics

[5] http://www.sysml.org/
[6] https://www.openmodelica.org/index.php/home/tools/134
[7] http://www.omg.org/mda/

and Section 5.4 introduces some guidelines related to model organisation.

5.1 Abstract Syntax

Three main constructs can be identified underpinning a spatial simulation: Spatial variables, Glocal variables and Animats. **Spatial** variables are spatial information layers that have some sort of impact on the dynamics of a simulation, e.g. slope that deters urban sprawl or biomass that feeds a wildfire. **Animat** is a term coined by Wilson [1991] signifying *artificial animal*; in this context it is used more widely, representing all spatial elements that change or induce change in their surroundings; examples are: fire (in a wildfire model), urban areas (in a sprawling model) or predators (in a population dynamics model). **Global** variables provide information that is constant across the space of simulation, such as wind direction in a wildfire model or economic trends in an urban development model. Another important sort of context variables are those that support Animat internal state. An Animat is composed by a set of **Attributes** that describe each instance at a certain moment in time.

The elements considered so far focus on the information needed to run a spatial simulation, but more is required to capture spatial dynamics, the way animats act and react to the environment has to be made explicit. This character of simulation is termed **Operation**. DSL3S proposes a set of just six predefined animat operations, intending to match the essential properties of an agent, as outlined by Franklin and Grasser [1997] (*autonomous*, *continuous*, *reactive*, *proactive* and *mobile*) with the core concepts found in Cellular Automata (*state*, *neighbourhood*, *transition rules* and *time*). In their seminal book, Epstein and Axtell [1996] conceive a considerably larger set of operations, including elaborate processes such as trade and cultural exchange. The option for a strict set of operations rests on three reasons: (i) to keep the language compact and easy to learn; (ii) more refined operations are less common in spatial simulation applications and can eventually be composed with these simpler primitives; and (iii) to insulate the user from technical implementation details in the choice between Cellular Automata and Agent-based models. These animat operations are:

- **Emerge**: sets the conditions under which a new instance of an animat can appear in the simulation, i.e., the act of "birth"; an example may be an urban development simulation where the emergence of new urban spots is possible in an area that meets a certain set of criteria, like distance to transport infrastructure or topography.
- **Move**: relates an animat with one or more spatial variables or with other animats determining the locations that are more or less favourable to be in.
- **Replicate**: captures operations where an animat replicates itself, such as an organism in a biological simulation reproducing a sibling.
- **Supply**: provides access to animat internal attributes, thus making resources or information available to other animats. It is the supply side of an interaction between animats.
- **Harvest**: an operation that allows an animat to collect resources or information from other elements in its neighbourhood; it may concern other animats, targeting attributes, or spatial variables. Between animats it is the demand side of an interaction, the counterpart of **Supply**. Examples may be wildfire consuming biomass or the seizure of resources from another animat as with a predator-prey simulation.
- **Perish**: defines the circumstances under which an animat may cease to exist during simulation; examples can be a biological animal starving or a fire extinguishing.

Figure 1 presents these key constructs in a conceptual model. A **Simulation** is composed by a set of **Spatial** and **Global** variables plus a set of **Animats**; the latter are composed by a set of **Attributes** and **Operations**, that determine how their internal state evolves. An animat acts through different types of **Operations**, that can induce changes on global and spatial variables, or be employed to interact with other animats. Different Animat configurations can be assigned to a Simulation, thus creating a different simulation scenario.

5.2 Concrete Syntax

The *DSL3S UML profile* gives body to the abstract syntax outlined above, with constructs defined as UML stereotypes. The stereotype `Simulation` is used to host definitions such as the spatial extent of simulation. It bonds together all the other elements, as an entry point to the simulation.

The stereotype `Global` is intended to be a scalar value that can vary with time. It can, for instance, be set randomly at simulation start and/or made to evolve randomly each time step. It can also be feed into the model as a predefined time-series, that may be an input from a text file.

The `Spatial` stereotype is essentially a stub for the input of geo-referenced data. Each instance corresponds

Fig. 1: The DSL3S meta-model.

to a spatial layer (either in raster or vector format) with the characteristic of having an unequivocal value for each location in space. This stereotype also provides means for the fully random generation of spatial variables, that may be useful for prototyping with synthetic scenarios.

The stereotype `Animat` is an aggregation of attributes existing at an identifiable location in space. The `Attribute` stereotype is a single characteristic of the `Animat`, representable by an atomic data type, such as an integer or a boolean (e.g. population in an urban development simulation). The initial number of animat instances of each type, and their spatial positioning, can be provided by a specific geo-referenced data set, such as a raster map. The values of `Attribute` elements can also be initialised with the same spatial data set, through its attribute table. These initial `Animat` and `Attribute` settings can also be randomly generated for simulations where it may apply.

The animat operations identified previously are also stereotypes in DSL3S; in detail:

- `Emerge`: this stereotype defines neighbourhood thresholds relative to spatial variables, or relative to other animat attributes, above which the emergence of a new animat becomes possible. When a new animat is created, its initial state is set according to the parameters set in the `Animat` class itself.
- `Move`: this stereotype provides properties to weight the relevance of each related class influencing the movement of an animat. For instance, in a predator-prey simulation the movement of a ”sheep” animat may be positively weighted in relation to a ”grass” `Spatial` layer and negatively weighted in relation to a ”wolf” animat.
- `Replicate`: this stereotype provides properties to set replication thresholds against the animat internal state. Impact on the reproducing animat and inheritance of attribute values to the new animat can also be modelled with specific properties. As with the `Emerge` operation, the initial state of a new animat resulting from a replication is set according to the properties set in the `Animat` class itself.
- `Supply`: together with `Harvest` this stereotype provides ways for anitams to exchange assets. It makes the information or resource held in a particular `Attribute` available to other animat. A limit may be set on the amount of this asset that another animat may acquire in each interaction.
- `Harvest`: this stereotype provides properties to parametrise how an interaction impacts the harvesting `Animat`. This is modelled with an harvest rate or harvest amount. A consumption rate of 100% may be used to model preying relationships, whereas 0% can be used to simply collect information on neighbouring animats and variables.
- `Perish`: This stereotype defines an interval of values relative to an `Attribute` element, determining the conditions for the existence of the `Animat` itself.

Other operation stereotypes can be added in the future if necessary; DSL3S is conceived to remain a language open to further extension.

5.3 Structural semantics

To properly define a DSL3S simulation a set of rules must be followed regarding the valid associations between the different language constructs. Table 1 synthesises these rules, indicating which relations are valid and their respective cardinalities. A more thorough description of these rules follows.

Each DSL3S model must contain exactly one `Simulation` construct. To it each `Animat`, `Global` or `Spatial` elements composing the model must be associated.

`Spatial` and `Global` variables represent passive constructs, but may appear associated with operation constructs, in such cases becoming sources of information and resources to `Animat` elements. As for `Attribute` constructs, they must always be associated to exactly one `Animat` (the owner).

An `Animat` aggregates `Attribute` elements, defining its internal composition. `Animats` do not link directly to any of the information constructs, `Spatial` or `Global`, neither to other `Animats`. All associations of an `Animat` with other elements of a simulation are made through its operation constructs.

A `Move` construct associates an `Animat` with other spatial objects. It can create a link to an `Attribute` or to a `Spatial` variable, quantifying propensity for movement. Beyond the link to the owner `Animat`, each `Move` construct must also link to exactly one other construct.

`Emerge` constructs are subject to rules similar to those applying to the `Move` operation, they must always link one `Animat` (its owner) with another construct in the model. Beyond `Attribute` and `Spatial` constructs, `Emerge` can also associate an `Animat` to a `Global` construct.

The `Supply` construct must always be associated to an `Attribute`, to which it provides access. It can then be associated to multiple `Harvest` constructs that access the resource or information supplied.

`Harvest` must also be always associated to an `Attribute` that stores the collected resource or information. On the other end it may associate to a single other construct: `Supply` (in case the harvested target is an `Animat`), `Global` or `Spatial`.

`Replicate` and `Perish` construct are simpler, since each must be linked to a sole `Attribute` construct, creating the boundary conditions for the respective operation. They can not be associated with any other construct, and thus each can only take part in a single association in the model.

5.4 Model Organisation - Views

Models built with DSL3S can become visually complex if a single diagram is used to represent all classes, properties and associations. To avoid such difficulties and provide a thorough structure for the development and presentation of models with the language, a multi-view approach is proposed. These views intend to display the model in such a way that each aspect of a simulation can be better presented in a specific diagram, namely the following: Simulation, Animat, Animat Interactions and Scenario views (see Figure 2).

The **Simulation View** contains the model settings and the participating variables. It includes the single `Simulation` construct plus the necessary `Global` and `Spatial` elements.

The **Animat View** provides a container where to define the structure of an animat. In this view an `Animat` and its belonging `Attribute` constructs should be present, plus any associations to `Spatial` or `Global` elements. This includes any operations linked to these elements: `Emerge`, `Move`, `Replicate` or `Perish`. A view of this kind per animat is recommended, thus visually encapsulating its configuration.

The **Animat Interaction View** is used to to describe operations between animats. It should contain all the `Supply` and `Harvest` constructs relating two (or more) animats, plus associated `Attribute` elements. `Move` operations relative to other animats may also be set in this view.

Lastly the **Scenario View**, is used to assign animats to a simulation. In this way, the designer may explore different animat configurations that can be used in different runs of a same simulation.

This multi-view structure is recommended, but does not have to be necessarily followed to develop a model with DSL3S. The user is free to use alternative organisations that may be considered more appropriate in specific cases.

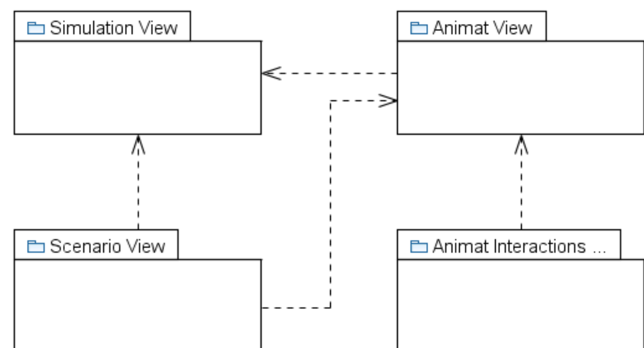

Fig. 2: DSL3S model views.

Table 1: Valid relationships in DSL3S with respective cardinalities.

	Simulation	Spatial	Global	Animat	Attribute	Emerge	Move	Replicate	Supply	Harvest	Perish
Simulation	-	0..N	0..N	0..N	-	-	-	-	-	-	-
Spatial	1	-	-	-	-	0..N	0..N	-	-	0..N	-
Global	1	-	-	-	-	0..N	-	-	-	0..N	-
Animat	1	-	-	-	0..N	0..N	0..N	-	-	-	-
Attribute	-	-	-	1	-	0..N	0..N	0..N	0..N	0..N	0..N
Emerge	-	0..1	0..1	1	0..1	-	-	-	-	-	-
Move	-	0..1	-	1	0..1	-	-	-	-	-	-
Replicate	-	-	-	-	1	-	-	-	-	-	-
Supply	-	-	-	-	1	-	-	-	-	0..N	-
Harvest	-	0..1	0..1	-	1	-	-	-	0..1	-	-
Perish	-	-	-	-	1	-	-	-	-	-	-

6 MDD3S - prototype implementation

"*Model Driven Development for Spatial Simulation Scenarios*" (MDD3S) is the name of the prototype framework that supports the DSL3S language. MDD3S relies solely on open source tools (see Figure 3):

(i) Papyrus - an Eclipse[8] add-on for UML modelling supporting the DSL3S UML profile;
(ii) Acceleo - another Eclipse add-on supporting the model-to-code transformation templates;
(iii) MASON - a *Program-level* spatial simulation framework used as a library by the code generated.

This section reviews some relevant aspects of these technologies in the scope of the MDD3S framework.

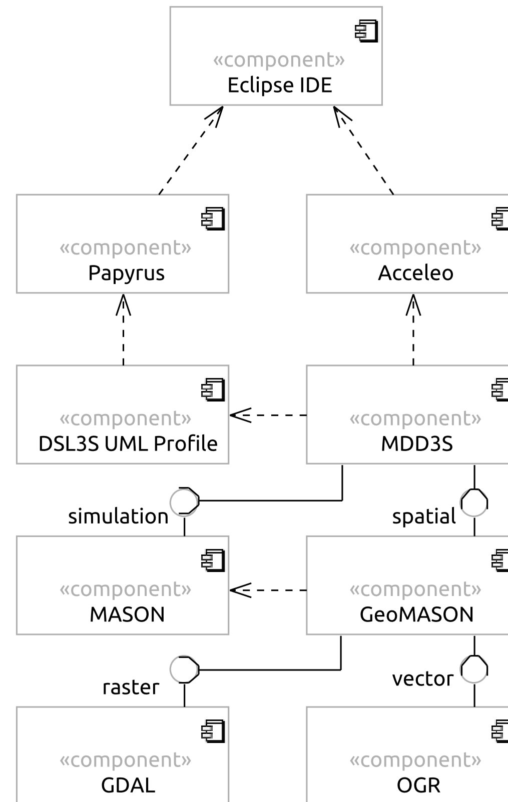

Fig. 3: The technologies used to implement MDD3S.

6.1 Papyrus

Papyrus[9] is an open source project started by the *Commissariat á l'Énergie Atomique* in France, with the aim of producing an advanced graphical editor for the UML language. It is based on the Eclipse Modelling Framework[10] (EMF), allowing the edition and visualisation of structured models defined with the XMI standard. It also provides a set of Java classes to facilitate model manipulation. Presently Papyrus is close to fully support version 2 of UML, bearing the development of *ad hoc* DSLs through the definition of UML profiles.

6.2 Acceleo

Acceleo[11] is an open source code generator created by the French company Obeo. It is also built on EMF, facilitating interoperability with several other EMF based

[8] http://www.eclipse.org/modeling
[9] http://www.eclipse.org/modeling/mdt/papyrus/
[10] http://www.eclipse.org/modeling/emf/
[11] http://www.acceleo.org/pages/introduction/en

Table 2: The Java service `hasLinkedStereotype` used in MDD3S to determine if a model element is linked to elements of specific type.

```
public boolean hasLinkedStereotype(
 Class c, String linkedStereotype) throws IOException
{
 EList<Association> associations = c.getAssociations();

 for (Association ass : associations)
 {
  EList<Element> elems = ass.getRelatedElements();
  for (Element elm : elems)
  {
   List<Stereotype> stereotypes = elm.getAppliedStereotypes();
   for (Stereotype stereotype : stereotypes)
    if (stereotype.getName().equals(linkedStereotype))
     return true;
   }
  }

 return false;
}
```

modelling tools. Acceleo interprets the MOF Model to Text Transformation language[12] (MOFM2T), also an OMG standard. Though not yet fully implementing MOFM2T, the model-to-code generators produced with Acceleo are today possibly the closest to the scheme proposed by the OMG.

The model-to-code transformation mechanism is based on special files called templates, which define the text output to produce from a graphical model. They are composed by regular text plus a series of annotations that are substituted by values and names of model elements at transformation time. Traditional computational operations such as branches or loops are also possible to include with specific annotations, producing more complex generators. Templates can be articulated through an inclusion mechanism, whereby a master template can make use of several other templates, creating a transformation chain. When fully developed, a transformation chain can be transformed into an independent plug-in for Eclipse, facilitating its portability and application.

Acceleo 3 fully supports transformations from models using UML profiles, identifying stereotypes applied on classes and providing access to its properties. The later is not based on MOFM2T, but provided by a service, essentially a Java method that browses through the UML object model associated with each class (example in Table 2).

When a transformation chain is applied on a model all its elements are run through the several templates declared in the master. Typically, the template file filters each element, generating code only for those with

[12] http://www.omg.org/spec/MOFM2T/1.0/

Table 3: The MDD3S template for the `Perish` stereotype; it parses an `Animat` element and successively iterates through each of the associated `Attribute` elements and to the `Perish` elements associated to these. `hasStereotype()`, `isNotNull()` and `getTaggedValue()` are external services.

Acceleo code template

```
[template public behavPerish(c : Class) ?
 (c.hasStereotype('Animat'))]
protected void perish(Sim sim)
{
[for (ass:Association | c.getAssociations())]
 [for (s:Element | ass.relatedElement) ]
  [let sClass: Class = s.oclAsType(Class)]
   [if (sClass.hasStereotype('Attribute'))]
    [for (assP:Association | sClass.getAssociations())]
     [for (p:Element | assP.relatedElement) ]
      [let pClass : Class = p.oclAsType(Class)]
       [if (pClass.isNotNull())]
        [if (pClass.hasStereotype('Perish'))]
         [if pClass.getTaggedValue(pClass,
            'Perish', 'upperThreshold').isNotNull()]
    if(attribute[sClass.name/] >= upperThresh[pClass.name/])
      sim.addTo[c.name/]Garbage(this);
         [/if]

         [if pClass.getTaggedValue(pClass,
            'Perish', 'lowerThreshold').isNotNull()]
    if(attribute[sClass.name/] <= lowerThresh[pClass.name/])
      sim.addTo[c.name/]Garbage(this);
         [/if]
        [/if]
       [/if]
      [/let]
     [/for]
    [/for]
   [/if]
  [/let]
 [/for]
[/for]
}
[/template]
```

Sample output

```
  protected void perish(Sim sim)
  {
    if(attributePreyEnergy >= upperThreshPreyPerish)
      sim.addToPreyGarbage(this);

    if(attributePreyEnergy <= lowerThreshPreyPerish)
      sim.addToPreyGarbage(this);
  }
```

a specific stereotype applied on. Such is the case with MDD3S, a template named `Simulation`, for example, generates the code for elements with the homonym stereotype applied. In most cases a template produces a text file, in MDD3S these are the Java classes that compose the end model. Alternatively, a template may simply generate a segment of code to be included in another file; in MDD3S the `Perish` template is an example, producing a method to be included in Java class generated for `Animat` type elements (Table 3).

6.3 MASON

MASON (acronym for Multi-Agent Simulator Of Neighbourhoods) aims to be a light-weight, highly-portable, multi-purpose agent-based modelling package [Luke et al., 2005]. MASON is a tool that in some aspects contrasts with earlier simulation packages like Swarm or RePAST that date back to the 1990s, following a strict object oriented philosophy from its very beginning. Its objects are architected in such a way that simulation models are completely isolated from visualisation and input/output mechanisms. MASON is fully written in Java and open source, producing programs that are highly portable, not only running alike, but also presenting identical results across different platforms. Comparative results have shown that MASON is likely the fastest of the main *Program-level* tools for spatial simulation [Railsback et al., 2006]. Supported by extensive documentation and a relevant community[13], MASON has slowly expanded its adoption.

GeoMason[14] is an extension that provides objects to deal specifically with geo-referenced data. Input and output functionality is available for both raster and vector datasets, relying on third party packages: the Java Topology Suite[15] for geometry manipulation, GeoTools[16] for vector formats input/output and GDAL[17] for raster formats.

Its light-weight infrastructure, extensive documentation, and ease of integration through Eclipse made MASON an obvious choice to support MDD3S.

7 Validation

The DSL3S UML profile and its accompanying MDD3S framework are publicly available at the code sharing platform GitHub[18]. Some examples are also available that showcase the usage of the language. In this section three of these illustrative simulations are discussed.

7.1 Simulation Model A – Predator-Prey

Predator-Prey simulations are one of the oldest applications of spatial simulation techniques [Dewdney, 1988], used to study population dynamics in the field of Biology. It usually features two animal species, where one feeds of the other; energy flows through the food chain in waves, whose period and amplitude are function of the growth rates of the several species.

[13] http://cs.gmu.edu/ eclab/projects/mason

[14] http://cs.gmu.edu/ eclab/projects/mason/extensions/geomason/

[15] http://www.vividsolutions.com/JTS/JTSHome.htm

[16] http://www.geotools.org/

[17] http://www.gdal.org/

[18] https://github.com/MDDLingo/DSL3S

7.1.1 The DSL3S model.

This example takes place in a synthetic plane of 100 by 100 abstract space units. There are three main elements to this simulation: a `Spatial` variable named *Pasture* and two animats: *Predator* and *Prey* (Figure 4). *Pasture* covers the whole simulation space and is initiated from a sample raster file that represents energy available at each space unit. This energy at each location increases at each time step, at a fixed rate, up to a defined limit.

Prey is an herbivore animat composed by a single `Attribute`: *PreyEnergy*. At simulation start a number of these animats are cast randomly across the simulation space, with its *Energy* attribute also randomly initialised. *PreyEnergy* declines steadily at each time step by a defined amount. A `Perish` operation attached to *Energy* sets a lower threshold below which the animat is discarded from the simulation. An `Harvest` operation parametrises the feeding act of *Prey* over *Pasture*; at each time step the animat can take all the *Pasture* energy available at the location it occupies into its own *PreyEnergy* attribute. Two `Move` operations relate *Prey* with both *Pasture* and *PredEnergy*, making it prefer locations with high *Pasture* energy and free of *Predator* instances. Finally, a `Replicate` operation sets a threshold above which the *Prey* can reproduce itself, as so the amount of energy passed on to the offspring in the process (the DSL3S view for *Prey* is shown in Figure 5).

Predator is a carnivore animat that shares many similarities with *Prey*. It also possesses a single attribute (*PredEnergy*) and its instances are created from an input vector layer, using the layer attribute table to initialise *Predator* attributes. Its energy also declines with time and a `Perish` operation determines when it ceases to exist. *Predator* feeds on *Prey*, according to an `Harvest` operation linking to a `Supply` operation associated with *PreyEnergy*. When a *Predator* feeds of a *Prey* it takes up all of its energy, triggering the later's `Perish` operation. A single `Move` operation links *Predator* again to the *Prey* energy attribute, this way compelling it to move towards locations where well nurtured *Prey* instances exist. A `Replicate` operation sets similar reproduction conditions to those for *Prey* (figure 5 presents the *Predator* view).

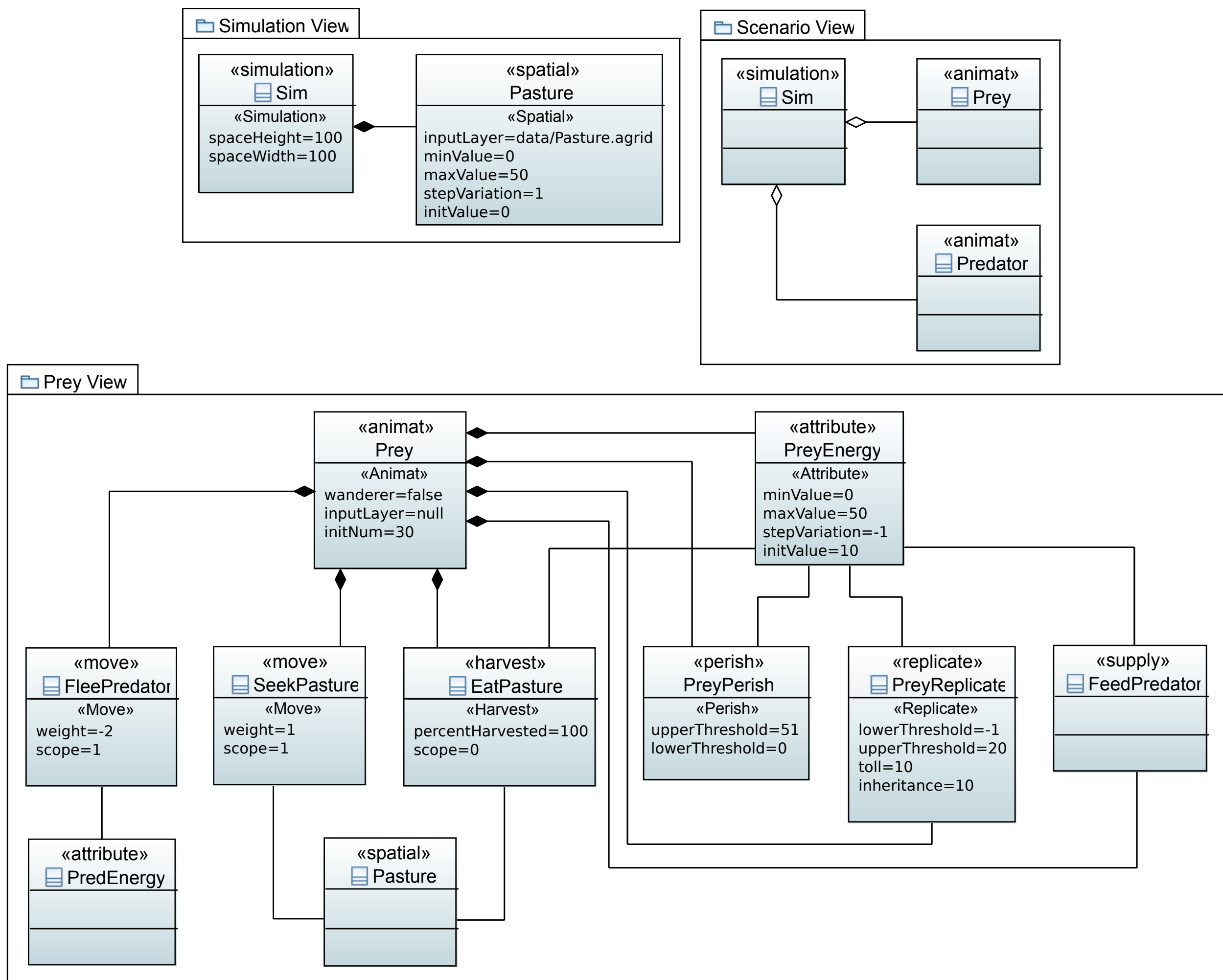

Fig. 4: Predator-Prey model in DSL3S; Simulation, Scenario and Prey Views.

7.1.2 The resulting application.

The simulation generated from this DSL3S model produces the typical population cycles seen in this type of models, as in the case of the historical WATOR model [Dewdney, 1988]. Figure 6 shows the simulation space during a sample run at time-steps 0, 30, and 90. *Prey* animats reproduce faster and thus dominate the space during the first time steps, producing an initial growth wave, reaping the fertile feedstock. In time, *Predator* animats feed of the excessive amount of *Prey* animats creating a new wave; this *Predator* wave clears some areas, fostering growth of the *Pasture* space variable in certain patches.

7.2 Simulation Model B – Forest Fire

Forest fire has been a classical application field for spatial simulation [Li and Magill, 2001], whereby the direction and intensity of fire at hypothetical locations is explored. The example here presented is rather simple, intended to illustrate other sorts of spatial dynamics possible to model with DSL3S.

7.2.1 The DSL3S model.

There are only two elements to this simulation, a `Spatial` variable for *Forest* and an `Animat` for *Fire*. *Forest* occupies the entire simulation space (100 by 100 units) and is initialised from a sample raster layer; it is not set to evolve with time.

Fire is composed of a single `Attribute`, registering the *Intensity* of each instance. At simulation start a pre-

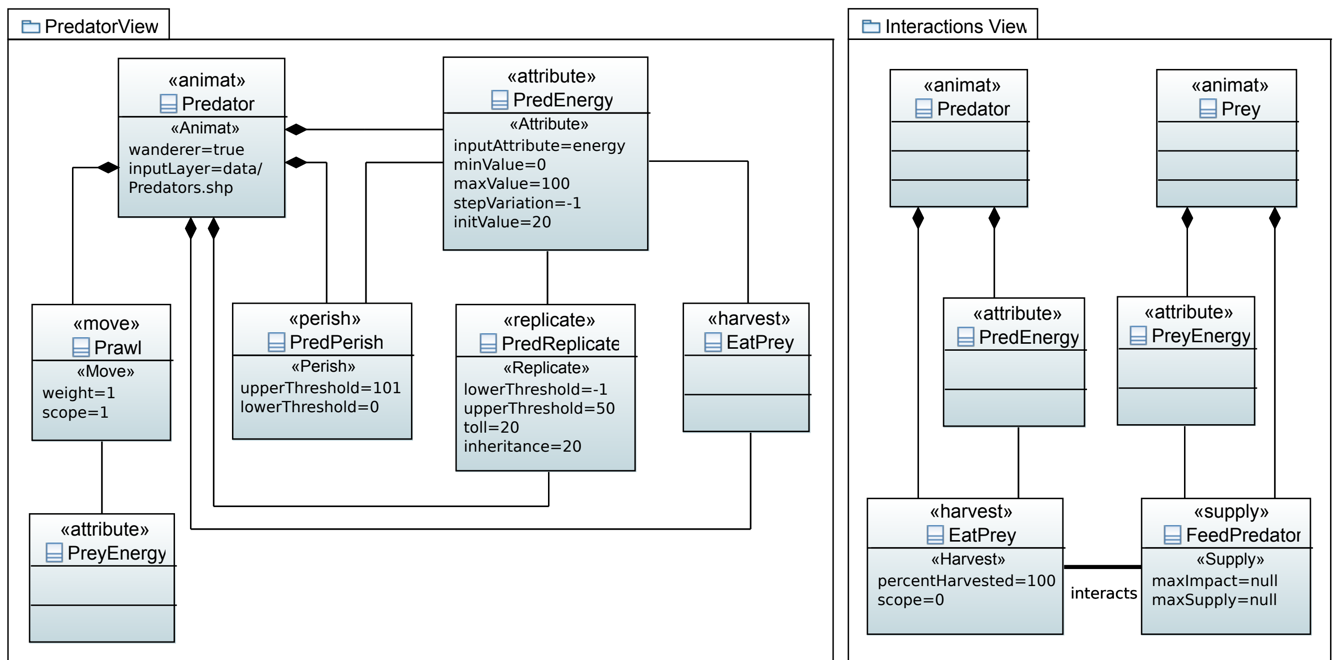

Fig. 5: Predator-Prey model in DSL3S; Predator and Interaction Views.

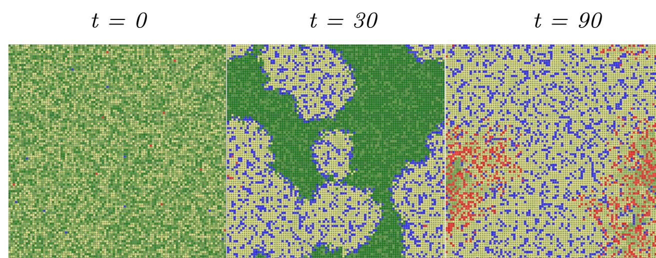

Fig. 6: A sample run of the Predator Prey DSL3S simulation; *Pasture* is portrayed with a yellow to green choropleth, *Prey* is portrayed in blue and *Predator* in red.

defined number of *Fire* instances is randomly cast in space with *Intensity* at its minimum value. An `Harvest` operation associates *Intensity* with *Forest*, defining the burning process, a constant depletion rate of the biomass existing at the location.

The amount of biomass burnt is transferred to the *Intensity* attribute, but at the beginning of each time step this variable is brought back again to its minimum; a `Perish` operation attached to *Intensity* guarantees that the animat is discarded if no biomass is left at the location (*Intensity* remains at the minimum). *Fire* is an animat that does not move, but it can spread to adjacent locations. This is modelled with `Emerge` operations, that link *Fire* to *Forest* and *Intensity*. The larger the amount of biomass in a location, and the more intense the fires burning in its neighbourhood, the higher the probability of a new *Fire* to emerge; the presence of a close by *Fire* animat is indispensable.

If the probabilities set for these `Emerge` operations are high enough, eventually most of the biomass burns down in this simulation. Still, it can be used to observe uneven fire spread patterns, following spatial patches with denser biomass. Figure 7 presents the diagrams defining the Fire simulation.

7.2.2 The resulting application.

Figure 8 portraits a run of the simulation generated from the Fire model described above. At time-step 0 three fire spots are randomly cast in the simulation space. With a relatively high probability of sprawling to adjacent cells, it slowly consumes the vegetation in every direction, especially towards locations with higher biomass density. The randomness of the emergence routine is apparent in the assorted locations left untouched after 100 time-steps.

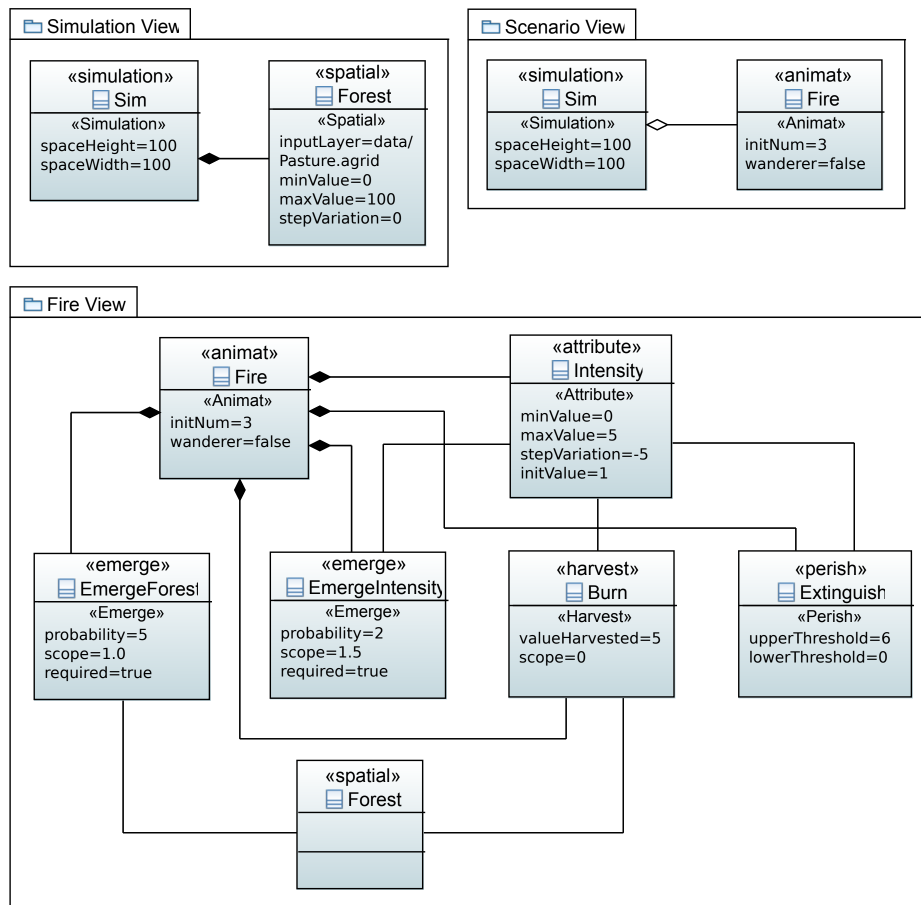

Fig. 7: Forest fire model in DSL3S.

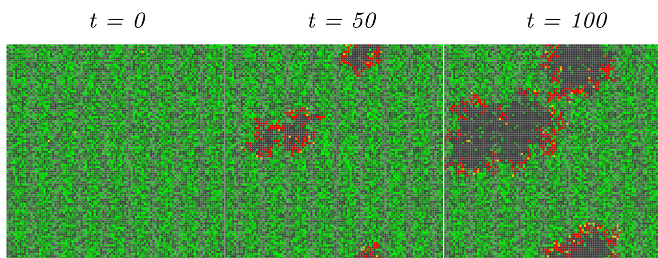

Fig. 8: A sample run of the Fire DSL3S simulation. *Fire* is represented with a choropleth from yellow (low intensity) to red (high intensity).

7.3 Simulation Model C – Urban Sprawl

Urban dynamics was another of the early application fields adopting spatial simulation techniques. The growth of cities is taken generally as an emergent process, bounded by spatial restrictions and enablers, by which the urban fabric sprawls. SLEUTH (Clarke et al. [1997]), a *Model-level* tool dating back to the 1990s, proved particularly successful in this domain and has been applied to varied geographic contexts.

7.3.1 The DSL3S model.

In this example space is vacant at simulation start and is progressively occupied by urban elements. The simulation is built around four elements: (i) an `Animat` named *Urbe* that possesses a single `Attribute`, storing its *Age*; (ii) a `Global` variable termed *Speed* that declines with time; (iii) a `Spatial` variable to input a vector layer with *Protected* areas, where urban growth is not possible; (iv) another `Spatial` variable that inputs a *Roads* layer, an enabler of urban sprawl. Figure 9 presents the full model in three views.

At simulation start a single *Urbe* animat is cast at random in the simulation space, it is not mobile and

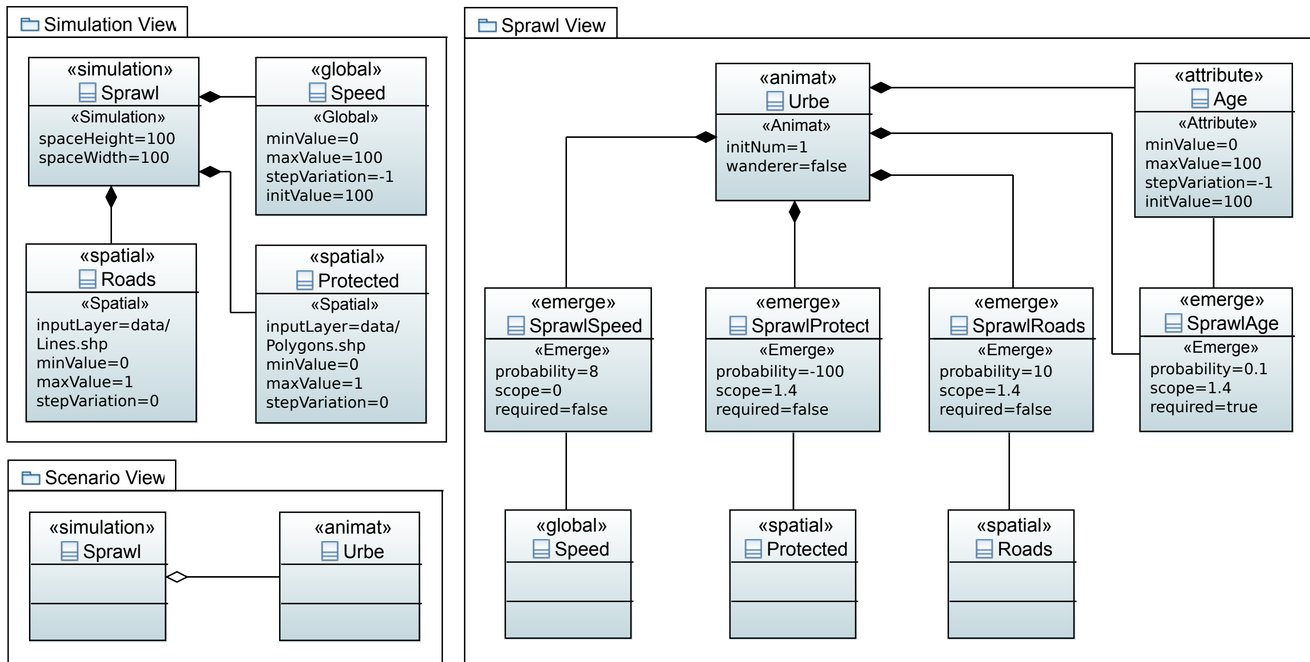

Fig. 9: Urban Sprawl model in DSL3S.

all the dynamics operates through related `Emerge` operations. In first place there is the *SprawlAge* operation that sets the probability of a new *Urbe* animat emerging nearby an existing *Urbe*; this operation is linked to the *Age* attribute, rendering emergence less probable near older urban areas. Also to constraint growth with time (mimicking diminishing capital investment) is the *SprawlSpeed* operation, linking to the *Speed* variable; as its internal value declines with time, it slows downs growth. *SprawlRoads* relates *Urbe* with *Roads*, increasing the probability of emergence around spatial features of this layer. In similar fashion, *SprawlProtect* sets the probability of emergence to zero (with a large negative weight) over spatial features in the *Protected* areas layer. The `Simulation` space is set to a grid of 100 by 100 cells, this way determining the space between adjacent emerging urban elements.

7.3.2 The resulting application.

Figure 10 shows the simulation space resulting from this model at time steps 0, 100 and 400. In the very beginning there is a single urban element, presented in red; in dark blue are shown *Protected* areas, while yellow lines portrait *Roads*. Development is swift in the beginning, with new urban elements emerging along road features; as they age, the colour of *Urbe* elements slowly fades to a light cyan. With time, sprawl slows down and a larger number of steps is required for changes in the urban fabric to become apparent. With the areas surrounding *Roads* taken, sprawl then turns inwards, but avoiding *Protected* polygons.

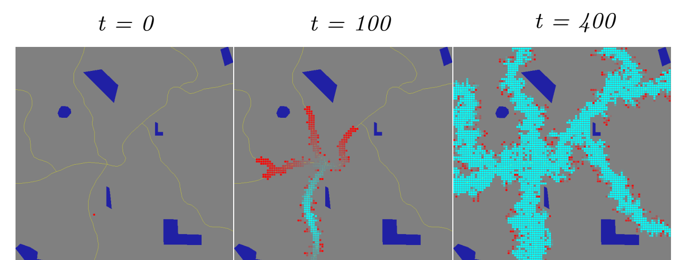

Fig. 10: A sample run of the Urban Sprawl DSL3S simulation; *Urbe* is portrayed with a red to cyan ramp, *Roads* are portrayed in yellow and *Protected* areas in dark blue.

8 Conclusion and Future Work

The application of spatial simulation techniques to the GIS domain remains today locked in the choice between versatile tools, and the option for ease of use with prebuilt models. The former require advanced programming skills, while the later impose relevant compromises of transparency and scope. A need exists for a higher level of development abstraction, that can also improve the documentation, readability and communication of models.

Several DSLs, and respective tools, such as NetLogo, Ocelet or GAML, were previously tried in this field, invariably producing declarative or functional languages, in some cases lacking a formal abstract syntax. Table 4 succinctly compares a selection of these with DSL3S. In essence, these efforts relying on textual languages end in the pitfalls identified by Selic [2008] regarding fourth generation languages: they struggle to hike the level of abstraction at which model development takes place. They also impose compromises with platform dependence and in some cases with weak spatial data support and interoperability.

This work proposes a different approach to this subject, applying the MDD philosophy. The end result is the DSL3S UML profile, that forms a graphical language, and its companion MDD3S framework, that involves a modelling and a model-to-code transformation infrastructure. These assets permit to translate a graphical and platform independent model produced with DSL3S into a coded simulation supported by a *Program-level* tool. In other fields this approach has proved capable of inducing faster development, reduce coding errors and improve model readability [Clark and Muller, 2012; Mohagheghi et al., 2013; Paige and Varró, 2012].

The application of DSL3S, and MDD tools in general, requires a certain degree of familiarity with graphical semantics (e.g. boxes and links) and modelling tools that may not be straightforward to all users. In spite of broad usage in computer science, languages such as UML do not yet feature in curricula of other technical disciplines. For this reason, DSL3S was conceived as a rather compact language, defining only eleven constructs, five structural and six operational. A set of expressive icons is also proposed, as so a structure of views that helps organising models developed with the language. This contrasts with AML, for instance, where the focus was mainly on defining a deeply detailed language, lacking any accompanying model-to-code transformation infrastructure.

The simplicity of DSL3S can be restrictive to some extent, but nevertheless, functionality provided by the MDD3S framework can already transform abstract models into executable code, allowing at least for prototyping. This article shows how DSL3S language elements can be combined to produce diverse simulations on different fields of application. To further ease the usage of DSL3S, a user manual has been created on-line [19] that is being expanded with examples and best practices. A series of tutorial videos will also be included.

Current development of the MDD3S framework relies on MASON, a modern Java library for spatial simulation. This option also guarantees interoperability with geographic data, namely through the GeoMASON extension. This framework is being developed on the Eclipse IDE, using the MDD ad-ons Papyrus (for UML modelling) and Acceleo (for model-to-code transformation). The code generated with MDD3S is relatively extensive *vis vis* the expected outcome from *ad hoc* development with a Program-level tool. MDD3S prizes simplicity and understandability over performance at this stage, a character of its purpose as a demonstrator prototype. If performance ever becomes a requirement model-to-code transformation templates can be optimised in that sense. Going further, transformation templates may even be developed to target a programming language closer to machine code such as C. Having a single abstract model producing different implementations relying on different code libraries is another distinctive advantage of the MDD approach.

In the near future, DSL3S will be further assessed through its application with real world scenarios. This iterative process will allow to understand how far it can go in its current form and if extensions are necessary.

[19] https://github.com/MDDLingo/DSL3S/wiki

Table 4: Comparison of several simulation DSLs with DSL3S.

	NetLogo	**SELES**	**MOBIDYC**	**Ocelet**	**GAML**	**AML**	**DSL3S**
Language							
Paradigm	functional	declarative	object oriented	declarative	declarative	UML profile	UML profile
Concept examples	Turtle Patch Link Observer Variable Agentset	Landscape Global Variable Event Agent	Agent Agent State Agent Behaviour Environment	Entity Service Relation Scenario Datafacer	Species Grid Reflex Init Task State	Entity Agent Environment Resource Structure Behaviour Attitude	Animat Attribute Operation Spatial Variable Global Variable
Abstract syntax	informal description	informal description	informal description	informal description	informal description	UML profile	UML profile
Concrete syntax	textual	textual	textual	textual	textual	graphical	graphical
Tool support							
Development environment	*ad-hoc* code editor	*ad-hoc* code editor	VisualWorks	Eclipse extension	GAMA	none	Eclipse extension
Platform	Java	unknown	Smalltalk	Java	Java	not implemented	multiple
Spatial data interoperability	only input	raster input/output	none	only input	input and output	not implemented	input and output
Usage							
Application domain	Multi-Domain	Landscape Dynamics	Population Dynamics	Landscape Dynamics	Spatial Simulation	Social Dynamics	Spatial Simulation
Coding skills	low to medium	none to medium	none to medium	low to medium	low to medium	not implemented	none

References

C. Atkinson and T. Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

M. Batty. *Cities and Complexity*. MIT Press, 2007.

M. Berryman. Review of software platforms for agent based models. *Science And Technology*, 2008.

A. Clark and P. Muller. Exploiting model driven technology: a tale of two startups. *Software & Systems Modeling*, 11(4):481–493, 2012.

K.C. Clarke, S. Hoppen, and L. Gaydos. A self-modifying cellular automaton model of historical urbanization in the San Francisco Bay area. *Environment and Planning B*, 24:247–261, 1997.

M. J. de Smith, M. F. Goodchild, and P. A. Longley. *Geospatial Analysis: A Comprehensive Guide to Principles, Techniques and Software Tools - Fifth Edition*, chapter Geocomputational methods and modeling, pages 625–672. Winchelsea Press, 2015.

L. M. de Sousa and A. R. Silva. Review of Spatial Simulation Tools for Geographic Information Systems. In *Proceedings of the Third International Conference on Advances in System Simulation (SIMUL 2011)*. ThinkMind, 2011.

P. Degenne, D. Lo Seen, D. Parigot, R. Forax, A. Tran, A. Ait Lahcen, O. Curé, and R. Jeansoulin. Design of a Domain Specific Language for modelling processes in landscapes. *Ecological Modelling*, 220:3527–3535, 2009.

A. K. Dewdney. *The Armchair Universe*, chapter Sharks and Fish on the Planet Wa-Tor, pages 239–251. W. H. Freeman, New York, 1988.

J. M. Epstein and R. Axtell. *Growing Artificial Societies*. MIT Press, 1996.

A. Fall and J. Fall. A domain-specific language for models of landscape dynamics. *Ecological Modelling*, 141: 1–18, 2001.

J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

D. F. Ferreira and A. R. Silva. RSLingo: An Information Extraction Approach toward Formal Requirements Specifications. In *Proceedings of Second IEEE International Workshop on Model-Driven Requirements Engineering (MoDRE)*. IEEE Computer Society, 2012.

R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

S. Franklin and A. Grasser. Is It an Agent or Just a Program? A Taxonomy for Autonomous Agents. In *Intelligent Agents III: Agent Theories, Architectures, and Languges*, pages 21–35, Berlin, 1997. Springer-Verlag.

H. Giese and S. Henkler. A survey of approaches for the visual model-driven development of next generation software-intensive systems. *Journal of Visual Languages and Computing*, 17(6):528–550, 2006.

V. Ginot, C. Le Page, and S. Souissi. A multi-agents architecture to enhance end-user individual based modelling. *Ecological Modelling*, 157:23–41, 2002.

A. Grignard, P. Taillandier, B. Gaudou, D. Vo, N. Q. Huynh, and A. Drogoul. Gama 1.6: Advancing the art of complex agent-based modeling and simulation. In G. Boella, E. Elkind, B. T. R. Savarimuthu, F. Dignum, and M. K. Purvis, editors, *PRIMA 2013: Principles and Practice of Multi-Agent Systems*, volume 8291 of *Lecture Notes in Computer Science*, pages 117–131. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-44926-0. doi: 10.1007/978-3-642-44927-7_9. URL `http://dx.doi.org/10.1007/978-3-642-44927-7_9`.

H. Iba. *Agent-Based Modeling and Simulation with Swarm*. Chapman and Hall/CRC, 2013.

M. Kraak and F. Ormeling. *Cartography: Visualization of Spatial Data*, chapter Data Acquisition. Prentice Hall; 3rd edition, 2009.

A. M. Law. *Simulation Modeling & Analysis, Fourth Ed.* McGraw-Hill, 2007.

X. Li and W. Magill. Modeling fire spread under environmental influence using a cellular automaton approach. *Complexity International*, 8:1–14, 2001.

S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. MASON: A Multi-Agent Simulation Environment. *Simulation: Transactions of the society for Modeling and Simulation International*, 82(7):517–527, 2005.

J. Merzenich and L. Frid. Projecting Landscape Conditions in Southern Utah Using VDDT. In M. Bevers and T. M. Barrett, editors, *Systems Analysis in Forest Resources: Proceedings of the 2003 Symposium*, pages 157–163, Portland, OR, 2005. U.S. Department of Agriculture, Forest Service, Pacific Northwest Research Station.

D. Mladenoff. LANDIS and forest landscape models. *Ecological Modelling*, 180(1):7 – 19, 2004.

P. Mohagheghi, W. Gilani, A. Stefanescu, M. Fernandez, B. Nordmoen, and M. Fritzsche. Where does model-driven engineering help? Experiences from three industrial cases. *Software & Systems Modeling*, 12(3):619–639, 2013.

B. Müller, S. Balbi, C. M. Buchmann, L. de Sousa, G. Dressler, J. Groeneveld, C. J. Klassert, Q. B. Le, J. D. A. Millington, H. Nolzen, D. C. Parker, J. G. Polhill, M. Schlüter, J. Schulze, N. Schwarz, Z. Sun, T. Patrick, and H. Weise. Standardised and transparent model descriptions for agent-based models: Current status and prospects. *Environmental Modelling & Software*, 55(0):156 – 163, 2014. ISSN 1364-8152. doi: http://dx.doi.org/10.1016/j.envsoft.2014.01.029. URL `http://www.sciencedirect.com/science/article/pii/S1364815214000395`.

M. J. North, T. R. Howe, N. T. Collier, and J. R. Vos. The Repast Simphony development environment. In *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, Chicago, USA, 2005.

OMG. UML 2.0 Specification. http://www.omg.org/spec/UML/2.0/, July 2005. Retrieved January 2011.

R. F. Paige and D. Varró. Lessons learned from building model-driven development tools. *Software & Systems Modeling*, 11(4):527–539, 2012.

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

A. Ribeiro and A. R. Silva. XIS-Mobile: A DSL for Mobile Applications. In *Proceedings of ACM SAC'2014 Conference*. ACM, 2014.

D. Samuelson and C. Macal. Agent-based simulation comes of age. *OR/MS Today*, 33(4):34–38, 2006.

J. Saraiva and A. R. Silva. Development of cms-based web-applications using a model-driven approach. In *Proceedings of the Simpsio para Estudantes de Doutoramento em Engenharia de Software (SEDES 2009, co-located with the ICSEA 2009)*. IEEE Computer Society, September 2009.

B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

B. Selic. Personal reflections on automation, programming culture, and model-based software engineering. *Automated Software Engineering*, 15(3-4):379–391, 2008.

A. R. Silva, J. Saraiva, D. Ferreira, R. Silva, and C. Videira. Integration of RE and MDE Paradigms: The ProjectIT Approach and Tools. *IET Software Journal*, 1(6):217–314, 2007a.

A. R. Silva, J. Saraiva, R. Silva, and C. Martins. XIS - UML Profile for eXtreme Modeling Interactive Systems. In *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007)*, Braga, Portugal, March 2007b. IEEE Computer Society.

R. Tobias and C. Hofmann. Evaluation of free Java-libraries for social-scientific agent based simulation. *Journal of Artificial Societies and Social Simulation*, 7(1), 2004.

I. Trencansky and R. Cervenka. Agent Modeling Language (AML): A Comprehensive Approach to Modeling MAS. *Informatica*, 29:391–400, 2005.

G. Weiss, editor. *Multiagents Systems: a Modern Approach to Distributed Artificial Intelligence.* MIT Press, 1999.

S. W. Wilson. The animat path to AI. In J. A. Meyer and S. Wilson, editors, *From Animals to Animats*, pages 15–21, Cambridge, MA, 1991. MIT Press.

A. Wuensche and M. Lesser. *The Global Dynamics of Cellular Automata*. Addison-Wesley, 1992.

Appendix A

This appendix details the properties of each DSL3S stereotype and how it influences the model-to-code transformation process.

A.1 `Simulation`

A.1.1 Properties

- *simulName* - A string with a title for the Simulation, to be presented in the user interface.
- *spaceHeight* - A numeric value indicating the height of the simulation space. Used to frame the class that hosts all animat instances during simulation.
- *spaceWidth* - A numeric value indicating the width of the simulation space; used as *spaceHeight*.

A.1.2 Artefacts

From this element two Java classes are generated: *Sim.java*, and *SimGUI.java* (where *Sim* is replaced with the content of the *simulName* property). The first contains all initialisation

routines, that create new animats and set up the simulation at its start; it also contains methods to discard animats during simulation time. The spatial data structures holding animats, `Spatial` variables plus `Global` variables are all properties of the *Sim.java* class too. *SimGUI.java* contains all properties required to present the simulation with a MASON GUI; it also initialises these properties.

A.2 `Animat`

A.2.1 Properties

- *inputFile* - Path to a spatial layer containing features setting the initial number and location of animats. Attributes of this layer can be used to initialise `Attributes` owned by this `Animat` element. Each feature in this spatial layer produces an singular `Animat` instance.
- *initNum* - The number of animats at simulation start. In case an input layer is not provided, this property is used to randomly cast animats in the simulation space.
- *wanderer* - Determines whether the animat should move randomly in case no `Move` stereotype is linked to it. Prevents an `Animat` instance from standing idle in an optimal location.
- *colourMin* - An RGB colour definition used to portrait the animat in the user interface. Defines a cloropleth together with *colourMax*, that is used in function of the values of one or more of the `Attribute` elements owned.
- *colourMax* - RGB colour defining the top end of the cloropleth used to portrait the `Animat`.

A.2.2 Artefacts

For each Animat class in the model a Java class with the same name is generated. It contains properties and methods to store and manage each of its `Attribute` elements, plus the MASON *step()* routine, containing the code to be executed at each simulation time step. A second Java class is also generated, again with the `Animat` name, but with the suffix *Portrayal*; as its name implies, it defines a cloropleth and methods to display the Animat in the GUI.

A.3 `Attribute`

A.3.1 Properties

- *attribute* - This property is used to initialise the `Attribute` instance with the value of the corresponding value in the attribute table of the spatial layer assigned to the owning `Animat` element.
- *minValue* - Minimum admissible value of the `Attribute`.
- *maxValue* - Maximum admissible value of the `Attribute`.
- *stepVariation* - Numerical value determining how the `Attribute` evolves with time. At each time step this value is summed to the `Attribute`.
- *display* - Boolean determining if the `Attribute` is used to portrait the owning `Animat`.
- *initValue* - Numerical value that initiates the `Attribute` (alternative to *initRandom* and *attribute*).
- *initRandom* - Boolean indicating if the Attribute is to be initiated with a random value (alternative to *initValue* and *attribute*).

A.3.2 Artefacts

A single Java class named *Variable.class* is generated, that serves as a generic encapsulation for atomic values used in the simulation. This class includes a set of properties (value, maximum, minimum and variation) and a set of corresponding

setter and getter methods. In the `Animat` Java class, a property of type *Variable* is generated for each owned `Attribute` element; the *initValue* and *initRandom* properties are included as static Java properties in the *Sim.class*.

A.4 `Spatial`

A.4.1 Properties

- *inputFile* - Path to a spatial layer initialising the `Spatial` element. In case it is a vector layer, each feature generates a `Spatial` instance; for raster layers each cell is transformed into a `Spatial` instance.
- *minValue* - Minimum admissible value for the variable.
- *maxValue* - Maximum admissible value for the variable.
- *stepVariation* - Numerical value determining how the variable evolves with time. At each time step this value is summed to the `Spatial` instance.
- *display* - Boolean determining if the `Spatial` variable is to be portrayed in the user interface. Since a `Spatial` variable can cover the entire simulation space, not all may be displayed.
- *initValue* - Numerical value that initiates the variable in all locations of the simulation space (alternative to *initRandom* and *inputFile*).
- *initRandom* - Boolean indicating if the variable is to be initiated with random values (alternative to *initValue* and *inputFile*).
- *colourMin* - RGB colour defining the lower end of the cloropleth used to portrait the variable.
- *colourMax* - RGB colour defining the top end of the cloropleth used to portrait the variable.

A.4.2 Artefacts

Elements of this type in the model produce a Java class with the same name. This class extends the *MasonGeometry* class from the GeoMASON library and includes a property of type *MasonGeometry*. A *Portrayal* class is also generated, with the same display functions as in the `Animat` case.

A.5 `Emerge`

A.5.1 Properties

- *probability* - Sets the probability of emergence of a new `Animat` in presence of the associated `Attribute` or `Spatial` variable.
- *scope* - Determines the radius of the neighbourhood to take into account in the emergence process.
- *required* - Boolean that sets the presence of the associated `Attribute` or `Spatial` variable as indispensable (or not) for the emergence of a new `Animat` instance.
- *lowerThreshold* - Lower value of the interval within which the associated `Attribute` or `Spatial` variable may trigger the emergence.
- *upperThreshold* - Upper value of the interval within which the associated `Attribute` or `Spatial` variable may trigger the emergence.

A.5.2 Artefacts

For each `Animat` in the model, a single Java class is produced, named after the `Animat` plus the suffix *Emerge*. The logic of every `Emerge` operation associated to the `Animat` is collected here. This Java class also has a *step()* method that is executed at each time step. It browses through the simulation space, at each location evaluating the probability of emergence. The

probabilities from each *Emerge* element are summed up, if the resulting value is above a random number generated between 0 and 100, then emergence takes place.

A.6 `Move`

A.6.1 Properties

– *weight* - Numerical value determining the weight of the associated element (`Attribute` or `Spatial`) in the movement process of the associated `Animat`.
– *scope* - Sets the radius of the neighbourhood to scout.

A.6.2 Artefacts

For each `Animat` a method named *move()* is generated containing the logic of all `Move` elements linked to it. This method is included in the corresponding `Animat` Java class and invoked in the *step()* method. At each time step the *move()* method searches for relevant instances within the neighbourhood(s) defined, collecting a set of candidate locations. The `Animat` moves into the location within the set of candidates with highest weight.

A.7 `Harvest`

A.7.1 Properties

– *percentHarvested* - Percentage of the harvested `Attribute` or `Spatial` variable to be taken (alternative to *valueHarvested*).
– *valueHarvested* - Exact quantity of the harvested `Attribute` or `Spatial` variable to be taken (alternative to *percentHarvested*).
– *maxIntake* - Sets a limit to the amount harvested from the associated `Attribute` or `Spatial`.
– *scope* - Sets the radius of the neighbourhood to take into account in the harvest process.

A.7.2 Artefacts

For each Animat a method named *harvest()* is generated containing the logic of all Harvest classes linked to the Animat in the model. It searches the within the neighbourhood(s) defined for relevant instances, invoking their harvest method counterparts as needed. This method is included in the corresponding Animat Java class and invoked in the *step()* method.

A.8 `Supply`

A.8.1 Properties

– *maxImpact* - Sets a limit to the amount supplied from the associated `Attribute`.

A.8.2 Artefacts

A method named *supplyAttribute()* is generated containing the logic of access to the *Variable* property of concerned `Attribute`. It is included in the respective `Animat` Java class.

A.9 `Replicate`

A.9.1 Properties

– *lowerThreshold* - Lower value of the interval within which the associated `Attribute` may trigger the replication process.

- *upperThreshold* - Upper value of the interval within which the associated `Attribute` may trigger the replication process.
- *toll* - Numerical value defining the impact of replication on the associated `Attribute` of the parent `Animat`.
- *inheritance* - Numerical value defining the amount transmitted to the associated `Attribute` from the parent to the child `Animat`.

A.9.2 Artefacts

For each Animat a method named *replicate()* is generated containing the logic of all `Replicate` classes linked to the `Animat` in the model. This method is included in the corresponding `Animat` Java class and invoked in the *step()* method.

A.10 `Perish`

A.10.1 Properties

- *lowerThreshold* - Lower threshold of the associated `Attribute` below which instances of the associated `Animat` are discarded.
- *upperThreshold* - Upper threshold of the associated `Attribute` below which instances of the associated `Animat` are discarded.

A.10.2 Artefacts

For each `Animat` Java class a method named *perish()* is generated containing the logic of all `Perish` classes linked in the model. It is invoked in the *step()* method a tests all the *Variable* type properties of concerned `Attribute` elements.