Ecological Informatics 27 (2015) 26–38

Contents lists available at ScienceDirect

Ecological Informatics

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

Domain-specific languages for ecological modelling

Niels Holst [a,*], Getachew F. Belete [b]

[a] Department of Agroecology, Aarhus University, Forsøgsvej 1, 4200 Slagelse, Denmark
[b] Department of Geo-information Processing, Twente University, Veenstraat 40, 7511 AS Enschede, Netherlands

ARTICLE INFO

Article history:
Received 16 February 2015
Accepted 24 February 2015
Available online 28 February 2015

Keywords:
Object-oriented
Component-based
Framework
Software design

ABSTRACT

The primary concern of an ecological modeller is to construct a model that is mathematically correct and that correctly represents the essence of a natural system. When models are published as software, it is moreover in the hope of capturing an audience who will use and appreciate the model. For that purpose, the model software must be provided with an intuitive, flexible and expressive user interface. A graphical user interface (GUI) is the commonly accepted norm but in this review we suggest, that a domain-specific language (DSL) in many cases would provide as good an interface as a GUI, or even better. We identified only 13 DSLs that have been used in ecological modelling, revealing a general ignorance of DSLs in the ecological modelling community. Moreover, most of these DSLs were not formulated for the ecological modelling domain but for the broader, generic modelling domain. We discuss how DSLs could possibly fill out a vacant niche in the dominant paradigm for ecological modelling, which is modular, object-oriented and often component-based. We conclude that ecological modelling would benefit from a wider appreciation of DSL methodology. Especially, there is a scope for new DSLs operating in the rich concepts of ecology, rather than in the bland concepts of modelling generics.

Contents

1. Introduction . . . 26
2. Earlier reviews . . . 27
3. Model building blocks . . . 27
4. The DSL niche . . . 28
5. DSL applications in ecological modelling . . . 29
6. Discussion . . . 30
Acknowledgements . . . 32
Appendix . . . 32
References . . . 37

1. Introduction

Ecological modellers have applied a variety of tools for model construction: general programming languages, e.g., Fortran, C++ and Java; mathematical software, e.g. Matlab (MathWorks, Natick, MA, USA) and R (R Development Core Team, 2014); and dedicated modelling software, e.g. STELLA (ISEE Systems, Lebanon, NH, USA) and Simile (Muetzelfeldt and Massheder, 2003). However, none of these tools constitute a domain-specific language (DSL). A DSL is a computer programming language of limited expressiveness focused at a particular problem domain (Fowler, 2011; Harvey, 2005). Thus an ecological model programmed in a DSL makes an effectively communicated statement about the ecological rationale and function of the model. Because of its sharp focus, a DSL does not provide the numerous capabilities of a general-purpose programming language. It just supports the minimum of features needed to support its domain. An appropriate DSL will facilitate quick and effective software development, yielding programs that are easy to understand and maintain. DSLs enable solutions to be expressed in the dialect and at the level of abstraction of the problem domain. Some DSLs might even be used by non-programmers (van Deursen et al., 2000).

New programming techniques are often taken up rather slowly, both by ecological modellers and by natural scientists in general (Derry, 1998; Merali, 2010). It is our hypothesis that ecological modellers, so far, have largely been unaware of DSL methodology. As an example, many readers proficient in R have already used DSLs unknowingly: the R packages *ggplot2* and *plyr* are both DSLs (Wickham, 2015); however, neither is for ecological modelling. In this review we explore the use of DSLs in ecological modelling and discuss how modellers could benefit from a wider application of DSLs.

* Corresponding author. Tel.: +45 22 28 33 40.
E-mail address: niels.holst@agrsci.dk (N. Holst).

http://dx.doi.org/10.1016/j.ecoinf.2015.02.005
1574-9541/

The choice of a modelling tool is naturally determined by habit. If a modeller has earlier experience with software for data analysis, it is convenient to use the same tool for rapid prototyping or even for the full implementation of a model. This may be the background for models developed in, for example, Matlab, R or spread sheets. A modeller who is also a teacher of modelling will likely be acquainted with graphic modelling tools, which give students a gentle entry to modelling. This results in models implemented in general modelling tools, such as STELLA. It is a matter of debate whether such graphical modelling tools are better suited for prototyping (Villa, 2001) than for serious modelling (Constanza and Voinov, 2003). Some modelling languages appear as general programming languages with simulation-specific features added. We find these languages too unconstrained to qualify as DSLs, admitting that the distinction is not clear-cut (cf. Fowler, 2011). Thus we have excluded DEVS (Zeigler, 1987), a successful, object-oriented language for discrete-event simulation models, and NetLogo (Tisue and Wilensky, 2004), a successful tool for individual-based modelling, from the review.

2. Earlier reviews

It is a long-standing goal to produce code that is flexible, modular and open for re-use, both in software engineering in general (Martin, 2009) and in ecological modelling in particular (Silvert, 1993). How to achieve this goal in ecological modelling has been the topic of several earlier reviews. Thus, Liu and Ashton (1995) and Peng (2000) reviewed the history of forest modelling, noting how the general evolution of software from the 1960s to the 1990s was expressed in the implementation and design of forest models. At first, models were programmed in procedural languages (e.g., Fortran, C) and were not designed for sharing or re-use. Then object-oriented languages (e.g., C++, Java) took over, and code re-usability, modularity and other aspects of 'clean code' (summarised by Martin, 2009) gained priority. Models also became increasingly user-friendly, as it became easier to develop dedicated graphical user interfaces (GUIs).

The ambition of developing a model, composed of re-usable building blocks, easily grows into the ambition of creating, not just another model, but a whole modelling tool for the domain in question, for example, forestry or hydrology. Argent (2004) lists the desired features of such a modelling tool; it should include a library of ready-to-use components, a development platform to construct new components from provided templates, a canvas on which to construct models from components, and a model execution environment. The canvas was envisaged as a GUI with drag-and-drop of model components. Argent (2004) did not mention DSLs as an alternative to the graphical canvas. Patrick Smith et al. (2005) displayed a similar bias towards graphical modelling tools; they considered modelling styles on a gradient from code-based to visual, ranging them from 'flexible and efficient' to 'user-friendly'. A DSL, possibly both code-based and user-friendly, was not considered.

To assess the user-friendliness of a modelling tool, or to develop a modelling tool with the aim of user-friendliness, the nature of the user group must be taken into account. The tool may be purely generic, targeting the modelling domain as such, or it may be focused on the domain for which models will be created. The distinction is important because the concepts of the tool should match the expertise of the users, either in the modelling domain or the applied domain (Harvey, 2005). An advantage of tools, focused at the applied domain, is that they make it easier and safer to construct models, because the components operate in the terms of the domain. When the meaning of components is obvious to the user, the components are more likely to be combined in a meaningful way (Adam et al., 2012; van Evert et al., 2005). Harvey (2005) saw benefits in using DSLs both in the modelling and applied domains, as long as they are not conflated. A well-designed DSL will by definition address a certain domain and serve a certain user group well.

In a practical comparison of modelling tools, Argent et al. (2006) set out to construct a spatially-explicit model of soil degradation using three different tools. Interestingly, they did not succeed in producing equivalent models. From this we conclude, that the choice of modelling tool is important for the resulting model, not only in form but in essence. There will be a limit to what a tool can conveniently express. The expressiveness of a modelling tool, DSL-based or not, depends on the nature of the building blocks that it supports. This we will consider next.

3. Model building blocks

Object-oriented design (OOD) design has been the dominant paradigm in ecological modelling since the 1990s, when Silvert's (1993) introductory paper set the milestone. Both Silvert (1993) and Reynolds and Acock (1997) argued that models should be constructed from modular, generic building blocks facilitating re-use. In OOD the building blocks are objects. To enable free combination of objects, they must match at the seams (have a common interface) and their binding must be loose, i.e. the Lego (tm) principle (Patrick Smith et al., 2005). Modern OOD offers techniques that allow both 'early' and 'late' binding of objects (through 'dependency injection', see Seemann, 2012). Thus, in a modelling context, one can imagine building blocks that only a modeller proficient in programming could compose to a working model (composition by coding, 'early binding'), or building blocks that a modeller could compose in a less demanding fashion, maybe with a visual tool or a DSL (composition by configuration, 'late binding').

Any OOD of some complexity will usually be arranged in a framework: 'A framework is a set of cooperating classes that makes up a reusable design for a specific class of software' (Gamma et al., 1995). In OOD any object belongs to a certain class which defines its functionality. A framework is organised as a hierarchy of classes with the most generic base class at the root. Modellers following the advice of Reynolds and Acock (1997) will have a root base class named *BuildingBlock* or something similar. In literature we found, for example, *BasicObject* (Larkin et al., 1988), *Population* (Silvert, 1993), *ModelComponent* (Baveco and Smeulders, 1994), *SimulationObject* (Sequeira et al., 1997), *Model* (Rahman et al., 2003), *ILinkableComponent* (Gijsbers and Gregersen, 2005) and *Component* (Holst, 2013; Moore et al., 2007).

Some differences in name-giving reflect the implementation language. Thus *ILinkableComponent* is an 'interface class', a type which is available in C# but not in C++, in which the same design is implemented as an 'abstract class' (e.g., *Component*). These base class names all reveal an intention of a highly generic modelling framework, except for *Population* which limits the scope to population dynamics.

A base class called *Model* indicates that any *Model* object is capable of running a simulation on its own, which indeed is the case for the *Model* objects of Rahman et al. (2003). Objects that can run as independent pieces of code have been called 'modules' (Jones et al., 2001) or 'components' (Papajorgji et al., 2004). We will use the term 'module'. Technically, a module can be an executable file or a dynamic link library, which in the right operating environment can be executed. Inter-module communication (maybe through 'services', see Papajorgji et al., 2004) then allows the composition of more complex models. Jones et al. (2001) and He et al. (2002) both advocated the use of modules, as they can communicate with each other through predefined interfaces to enable the joint action of models residing on different computing platforms. The Common Component Architecture (CCA, 2014) forms the basis for many model integration tools (see Peckham et al., 2013), in which the building blocks consist of whole, working models of various origin.

Whether model building blocks are supplied as a framework of classes or as a library of modules, a specific model is constructed by combining and configuring chosen blocks. Commonly, the design allows super-blocks to be combined from other blocks. The new super-block again can function as a block (a 'composite pattern', see Gamma et al., 1995). As an example, a framework providing the classes *Rotation*, *Crop* and *Organ* can be used to construct a model with an object *wheat* of class *Crop*, which has inside the objects *root*, *stem*, *leaf* and *ear* all of class *Organ*. With a similar object *maize* of the *Crop* class, a *conventional* object of class *Rotation* could hold the objects *wheat* and *maize*. In the context of

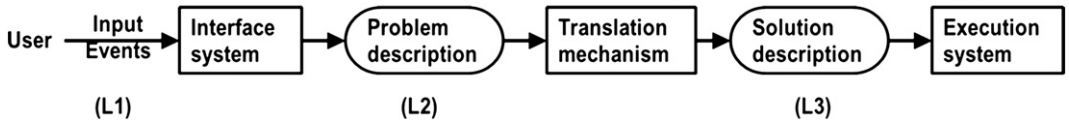

Fig. 1. Elements of a modelling system. L1–L3 denote formal languages (explicit or implied). After Robertson et al., 1989.

modules, one can imagine a *Weather* module and a *Crop* module, developed by separate teams, maybe even using different frameworks. The modules themselves consist of objects but they are encapsulated by the module, and we are relieved from dealing with this detail at the module level. The modules can run independently, the *Crop* module defaulting to a standard climate. Or they can run together, the *Weather* module providing input to the *Crop* module. One can imagine a library holding different versions of *Weather* and *Crop* modules, together with other modules. This library could serve as a palette of modelling building blocks, just like the framework of classes above, although the level of abstraction and the underlying software implementation would differ.

Model building blocks commonly have an interface consisting of ports for inputs and outputs (e.g. Maxwell and Costanza, 1997). Inputs may be divided further into parameters (providing fixed values) and input variables (providing dynamic values). During model configuration, the modeller sets parameter values and connects outputs to inputs as needed. This may lead to complex routes of information exchange between model building blocks, but just like a programmer, who works on a source code base counting millions of lines, does not need to understand the whole system in detail, likewise a modeller only need to contemplate the local interactions between building blocks. What makes this possible is 'clean code' (Martin, 2009) which divide concerns between objects and between modules equipped with well-defined interfaces. Peckham et al. (2013) advocated auto-connecting building blocks to assist the user.

The foremost characteristic of a model building block is its behaviour. It maintains an internal state which it keeps updated according to its inputs. The internal state is inaccessible, except through defined output ports. The behaviour of a building block is defined by the algorithm that updates its internal state. The algorithm is implemented in building block 'functions', also called 'methods'. These are defined strictly in many modelling frameworks by 'abstract methods' of the framework base class. Common abstract methods are *reset* or *init*, to initialise the object state before the simulation begins, and *update* or *run* to update the state according to current inputs (e.g., Holst, 2013; Kralisch and Krause, 2006; Peckham et al., 2013). In other designs, often called 'component-based' or 'service-oriented', methods can be more loosely defined and made available through OOD techniques known as 'introspection' or 'reflection' (Argent, 2007; Rahman et al., 2004). Some modellers prefer the freedom of component-based modelling (e.g., Moore et al., 2007), while others (e.g., Chabrier et al., 2007) prefer the stricter formalism imposed by a framework. Frameworks, that are well-defined for a domain, provide the modeller with the exact degree of flexibility needed to support the domain and protects him from irrelevant design decisions but, then again, such frameworks are exceedingly difficult to design (Gamma et al., 1995; Harvey, 2005).

4. The DSL niche

Before we consider how a DSL could help the modeller, we must consider where it would fit into the modelling process. If we think about how, in general, information can enter and be interpreted by a modelling system (Robertson et al., 1989) then, in a first step, the user provides 'input events' to an 'interface system' (Fig. 1). Common input events are clicked menu items, filled-in dialogue boxes, user-drawn model diagrams and typed-in scripts. These inputs are turned into a formal 'problem description' by the interface system. The 'translation mechanism' reads the problem description and creates the 'solution description' which is taken as an input by the 'execution system' to carry out the actual simulation.

To Robertson et al. (1989), the ideal input should take the form of purely ecological statements, which would be stored as a formal problem description, that an intelligent translation mechanism would turn into a solution description, digestible by the execution system. Harvey (2005) points out, that 'every modelling system implements one or more formal, computer-based languages'. Often these languages can be implicit as in the elements of a graphical user interface (GUI). In Fig. 1 we recognise up to three languages: for the input events (L1), for the problem description (L2) and for the solution description (L3). For instance, the modelling system could capture user gestures (L1), store them in an XML file (L2) and translate this into Java code (L3) entering the execution system. Or, the user could use a text editor to enter DSL code (L1), which would be stored directly as the problem description (L2), which would again be equivalent to the solution description (L3), which would enter the execution system directly. The second example demonstrates the simplicity of a pure DSL approach; only one language is needed to formulate and execute the model (L1 = L2 = L3).

The model build blocks, which the user is manipulating by input events, are not evident in Fig. 1 but, obviously, some parts of the software that implements the modelling system (interface system, translation mechanism and execution system) must be operating in terms of these building blocks, at least the user's interface system. To assist modeller creativity, L1 should use building blocks conceptualised in the application domain, but the modelling system must at some level be coded in a general programming language. Domain concepts may not necessarily spill through to all coding levels, but certainly it makes the coding more transparent if, for example, building blocks called *Plant* and *Organ* can be recognised in the coding as the classes *Plant* and *Organ* (Chabrier et al., 2007).

Modelling software often has a segregated user interface (Fig. 2): a development environment, in which new model building blocks are constructed, and a modelling canvas, on which building blocks can be placed

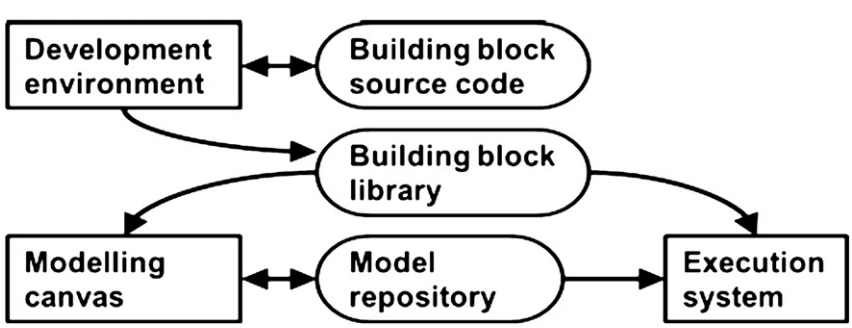

Fig. 2. The development and use of building blocks in a modelling system. 'Building blocks' are a joint term for objects, classes, modules and components.

to compose multi-block models (Argent, 2004). When new building blocks are ready for use, they are committed to a building block library. This library defines the palette of building blocks available for the modelling canvas and also provides the building block code needed by the execution system to carry out model simulations (Fig. 2). The execution system may need a translation mechanism as in Fig. 1 depending on the nature of the building block library and the model repository.

The designer of a modelling system (Fig. 2) will have to choose which features can be fulfilled by standard software, and which would have to be implemented as dedicated software. For instance, the development environment could be provided by a general programming environment, and the modelling canvas could be a standard text editor to compose models in a DSL. Only the execution system would then need to be implemented as dedicated software — to accept models and building blocks as inputs, carry out the execution and return the requested output.

5. DSL applications in ecological modelling

Our literature survey of DSLs applied in ecological modelling was not straightforward; only few modellers have been aware of the DSL concept. Therefore the use of DSLs has been mostly implicit. A common hiding place for unacknowledged DSLs was in the so-called 'configuration files', which provide information on model configuration, parameter values and options. These de facto DSLs have been noticed before both in ecological modelling (Harvey, 2005) and in software in general (Fowler, 2011). The following exposition gives a chronological overview of the roles played by DSLs in ecological modelling (Table 1). Code examples to give the flavour of each DSL are provided in the Appendix, Listings 1 to 13.

MOSES (Listing 1) was developed by Wenzel (1992) as a formal language which allowed 'a hierarchical structuring of models out of autonomous partial models residing in a model bank'. The basic building blocks were ecological processes of 20 different kinds. MOSES scripts were read by an execution system written in Fortran. Even though the building blocks were referred to as 'generic objects', MOSES was not fully object-oriented. The author seems to have realised that the terse syntax of MOSES was challenging (Listing 1), as he planned to supplement MOSES with a GUI to help the user construct models. In retrospect, as a DSL, MOSES does seem rather inconvenient. The same can be said of Object-Z, the mathematically oriented language used by Durnota (1994) to specify ecological interactions with the prospect of later (but never realised) implementation in an object-oriented language; a domain-specific user interface was suggested.

The OOMP framework for population dynamics modelling (Holst et al., 1997) reads a script (Listing 2) defining which objects to create; parameter values are supplied in a separate file. Considered as a DSL, the script has a simple, yet unpolished design. Objects present populations (of the *Model* base class) and trophic interactions (of the *Link* base class). Keywords 'consumes' and 'infects' (Listing 2) declare which kind of *Link* objects to create.

SML (Maxwell and Costanza, 1997) is a full-featured DSL for spatial ecological–economical modelling. The authors did not call it a DSL, most likely because the concept was not widely used at the time. An SML model consists of modules which are interfaced through inputs and outputs. The modules can be nested to form a hierarchy, which sets scope rules for which input–output connections are possible. An SML script (Listing 3) consists of a module declaration, which contains equations for variables and definitions of outputs. Variables may serve different purposes set by modifiers, such as 'state', 'flux' or 'input'. The latter allows input from other modules. To aid users, the accompanying software allows the import of models written in STELLA, which has a friendly GUI, which are then translated into SML. SML scripts are converted by the Spatial Model Engine (SME) into C++ code, which in a final step is compiled and executed.

MickL is an object-oriented C-like language used to code models for OME, the Open Modelling Engine (Reed et al., 1999). By way of the OME GUI, models are written in MickL, as equations that take inputs and parameters, to produce outputs. Variable names beginning with an uppercase letter can be read and used as inputs by other models, while other variables are local to the model (Listing 4). Both models and equations are loosely bound, so that a simulation can be composed as a hierarchy of both models and equations. The first version of OME read and interpreted MickL code but later versions (Rahman et al., 2004) contained a compiler which produced machine code enabling much faster execution of MickL code. The difficulties of maintaining such a low-level, compiled language are revealed by the later observation of Argent (2007), that MickL is a 'largely un-documented language'.

IMT, the Integrated Modelling Tookit (Villa, 2001), is a software designed to integrate landscape models from modules which may be of diverse origin. Similar to SML, models in IMT are composed as a hierarchy of modules. Scripts are written in XML code which in some case shares similarities with SML; for example when modules are defined in terms of 'stocks' (i.e., state variables) with input and output fluxes, to specify a differential equation model (Listing 5). The author describes how other types of modules can be defined to allow integration of existing models; an XML script for each module provides meta-data allowing modules to cooperate.

SELES (Fall and Fall, 2001) seems to be the first ecological modelling tool, in which the authors were aware of the DSL methodology and strived to meet all criteria of a proper DSL. SELES defines a language for landscape modelling, which comes with a discrete-event simulation engine and a GUI to guide model development and display model output. SELES targets the same domain as SML. What sets SELES apart is its advanced modelling of events spreading in the landscape. The coding example (Listing 6), however, has been chosen for brevity and has a simple event structure. Like SML, the SELES language combines declarative

Table 1
Domain-specific languages (DSLs) in ecological modelling. No.: Listing in Appendix. DSL: Name of DSL or modelling tool. Code: Implementation language for building blocks or modelling system. Abbreviations: Y = Yes and N = No. O = object-oriented and M = module-based. Agro-env. = Agro-environment. Pop.dyn = Population dynamics. UniSim = Universal Simulator.

No	DSL	Authors	Year	Application		XML-based	Building blocks			System	
				Topic	Spatial		Type	Nested	Code	GUI	Code
1	MOSES	Wenzel	1992	Ecology	N	N	O	Y	Fortran	N	Fortran
2	OOMP	Holst et al.	1997	Pop.dyn.	N	N	O	N	C++	N	C++
3	SML	Maxwell and Costanza	1997	Landscape	Y	N	M	Y	any	STELLA	C++
4	MickL	Reed et al.	1999	Hydrology	Y	N	O	Y	MickL	Canvas	C++
5	IMT	Villa	2001	Ecology	Y	Y	M	Y	any	N	?
6	SELES	Fall and Fall	2001	Landscape	Y	N	O	Y	SELES	Canvas	?
7	FarmSim	Good	2005	Agronomy	N	Y	O	Y	C#	End-user	C#
8	JAMS	Kralisch and Krause	2006	Hydrology	Y	Y	O	Y	Java	N	Java
9	CMP	Moore et al.	2007	Agro-env.	N	Y	M	Y	any	N	?
10	Ocelet	Degenne et al.	2009	Landscape	Y	N	M	N	Ocelet	Eclipse	Java
11	OMS3	David et al.	2012	Hydrology	Y	N	M	N	Java	N	Groovy
12	FlexSem	Larsen et al.	2013	Estuaries	Y	Y	O	N	XML	N	C++
13	UniSim	Holst	2013	Ecology	N	Y	O	Y	C++	N	C++

with procedural code (Listings 3 and 6). This means that the DSL is used not simply to declare, how the model is composed, but also to define some of its behaviour. Only common functionality, such as differential equations integration, is implicit and taken care of by the execution engine.

Ginot et al. (2002) defined 25 'primitives' for agent-based models and created a GUI to combine these primitives into 'tasks' to let the user define a model. They envisioned that the primitives could form the basis of a general 'platform-independent language' for agent-based modelling. Thus they laid out a basis for a DSL but did not implement it.

FarmSim (Good, 2005) is an object-oriented farm model which takes XML files as input for model configuration, and which also produces XML output as a simulation result. The hierarchical structure of XML, with the obligatory root node *farm* (Listing 7), is translated directly into a hierarchy of model objects by the FarmSim execution system. The authors note how the separation of input/output logic from model logic made the software development process more manageable. Thus the end-user GUI to handle XML input/output could be developed independently of model development.

L1 (Gaucherel et al., 2006) promises to be a DSL for modelling rural landscapes with a special focus on modelling the effects of farmer activities on landscape structure. However, the paper does not succeed to present a DSL, hence the status of L1 remains uncertain.

The development of the JAMS framework was targeted at hydrological modelling (Kralisch and Krause, 2006) but its design is of more generic nature. JAMS building blocks are 'components' configured via an XML script, which also provides concepts for 'spatial context' and 'temporal context' which defines the scheduling of component updates. Both components and contexts can be nested. Components are programmed in Java and have a simple interface of three virtual methods, *init*, *run* and *cleanup*, that can be specialised for each kind of model. The *run* method updates the component according to its context. Listing 8 shows how a component is set up in JAMS XML. The input *tmean* is fetched from the *TmeanDataReader* component and similarly for the *rhum* input. The component has one output called *vpd*. This shows the Java code which computes the output from the two inputs:

```
public void run() {
    double esT = 0.6108 * Math.exp((17.27 * tmean.getValue())/
    (237.3 + tmean.getValue()));
    double ea = esT * rhum.getValue() / 100.0;
    vpd.setValue(esT − ea);
}
```

Notice the clean uncoupling of code (cf. Martin, 2009); the Java code is ignorant of where the *tmean* and *rhum* values come from.

The Common Modeling Protocol (CMP) for agro-ecological modelling was claimed to be 'distinguished from existing simulation frameworks by taking an explicitly hierarchical view' (Moore et al., 2007). However, in light of the models reviewed above, the hierarchical stance was far from particular to CMP. In CMP (Listing 9) the user configures the model building blocks in an XML file, just like in IMT (Listing 5) and FarmSim (Listing 7). The domain of CMP is the composition of models from existing modules. Each module is referenced by the name of the executable file and provided with the initial values of its state variables (Listing 9). The simulation is carried out steered by events, and the modules communicate through messages. The modules themselves can be programmed in any language, as long as they obey the CMP message interface. The authors note that CMP is best suited for applications with a low frequency of inter-module messaging, because the messaging slows down model execution.

Ocelet (Degenne et al., 2009, 2010) is a DSL aimed at spatially-explicit ecological modelling. Ocelet models are composed of building blocks called 'entities' which provide an interface of services, and which can be hierarchically nested. Entities are connected through 'relations', which are not mere data wires but specific, information-bearing actions. Thus *Grazing* could define a relation between any two entities providing the *grass* and *herbivore* interface (Listing 10). The 'scenario' concept of Ocelet provides space and time context for the simulation to be carried out, similar to the 'contexts' of the JAMS framework. A translation mechanism turns Ocelet code into Java code, which is subsequently taken up by the Ocelet execution system. Ocelet code is entered by way of the Eclipse environment (Eclipse, 2014) tailored with an Ocelet plug-in.

David et al. (2012) created a family of DSLs for Object Modeling System 3 (OMS3). OMS3 works as an execution system which is able to interpret models configured with one of the OMS3 DSL dialects. OMS3 was implemented in the Groovy language — with extensions to ease the use of DSLs (Dearle, 2010). A DSL script tells OMS3 which components take part in a model, and which links exist between component output and input 'fields' (as in JAMS, Listing 8). In addition, parameter values and other input data can be set (Listing 11). In the component source code, variables can be declared as inputs or outputs using *@In* and *@Out* annotations, which allows them to be linked in the DSL code. In the examples provided (David et al., 2012), components are simple, i.e. not arranged in a composite structure as in the majority of DSLs (Table 1). This most likely reflects that composite components were not needed in these applications.

Larsen et al. (2013) presented Flexsem, a generic 3-D spatial model applied to estuarine modelling. A Flexsem model is specified by an XML configuration file. The XML is not purely declarative, as it contains procedural code in the form of equations (as in IMT, Listing 5), in addition to model composition and parameter values (Listing 12). The XML script is interpreted and executed by an application written in C++ in an environment that offers parallel, multi-processor execution.

Universal Simulator (Holst, 2013) and its precursor WeedML (Holst, 2010) use XML files to configure building blocks ('models') which can be nested (Listing 13). Universal Simulator comes with an execution system written in C++ which interprets and executes the XML scripts. Models are written in C++ and compiled to libraries ('plug-ins') which define the vocabulary of model classes available in the XML scripts. Models are defined by seven virtual functions, three of them (*initialize*, *update*, *cleanup*) play the same role as those of JAMS (*init*, *run*, *cleanup*) (Kralisch and Krause, 2006). Models have 'parameters' given default values in the C++ source code, which in the XML file can be overridden with fixed values ('value' attribute) or with values referenced from the output of other models ('ref' attribute) (Listing 13).

It is interesting to compare how data wires, from a source model's output to a target model's input are set up. In Universal Simulator (Listing 13), the reference to the source follows the path syntax known from file systems: '..' means one level up, '.' means this level. The path from one model to another is followed by the name of the output variable in brackets. In JAMS (Listing 8), the 'provider' attribute refers to the source (without a path so it must be unique) and the 'value' attribute specifies the output variable. In OMS3 (Listing 11), the data wires are not declared in each target component, as in Universal Simulator and JAMS, but are all declared in the 'connect' element in source-target pairs. Like in JAMS, references are simple (i.e., not paths), so it seems that component names must be unique. These differences in the DSLs likely reflect differences between the application domains.

6. Discussion

We identified 13 DSLs (Table 1) developed for the ecological domain; however, most of them are of much more generic nature. In their core concepts, they address the modelling domain as such, rather than any specific application. Within the modelling domain, most of the DSLs follow the stocks and fluxes paradigm, in which model building blocks accept inputs, update their internal state and produce outputs. However, what delineates the actual domain is the library of pre-fabricated building blocks that comes together with the DSL. A framework-based DSL with a base class called *Model* could in principle model anything, but if all the available models address hydrology, this becomes the de facto

domain of the language — until maybe someday somebody develops building blocks for another domain with that DSL. This process, of a DSL growing into its domain with time, was acknowledged by Degenne et al. (2009) (calling building blocks 'primitives'): 'We expect that a set of most useful types of primitives will emerge, from which modellers would pick and adapt to their case studies. This also implies that primitives building would always be part of the modelling exercise'. As a safeguard, DSL modellers will therefore opt to make the DSL itself extendible, so that unforeseen needs can be met as they emerge. How extensible the DSL is, will depend on how easy it is to extend the execution system with new functionalities that can be invoked by the DSL, e.g., through 'plugins' (Holst, 2013), a design well-known from R which can be extended with 'packages' (R Development Core Team, 2014).

Half of the DSLs are based on XML, the syntax of which is used to compose models from available building blocks. XML does seem a natural choice because it offers a direct syntax to form building block hierarchies. This feature is utilised by all the XML-based DSLs, except FlexSem. FlexSem also diverge, by defining building block behaviour (in the form of equations) in XML, whereas the others used XML only to declare model composition. Parameter values and connections between building block outputs and inputs were often included in the XML as well. Yet, in spite of the popularity of XML as a medium for DSLs, it is far from the DSL ideal. XML was invented as a data exchange file format, not as a format for the man-machine interface. Its proper usage is exemplified by SBML, an XML dialect for systems biology: 'Note that biologists and other software users are not intended to write their models in SBML by hand—it is the software tools that read and write the format' (Hucka et al., 2003). Likewise, Fowler (2011) acknowledges that XML can be used to construct DSLs but also argues heavily against it. XML syntax is burdened by details that carry no domain-specific meaning and thus works against the clarity that a DSL should offer. With modern tools available to construct parsers and interpreters (Parr, 2009), there is really no excuse for DSL developers not to construct their own, simple and clear DSL syntax. A simple, first solution could be to develop a better DSL for each XML-based DSL. For the implementation, one would only need to write software to translate the new DSL into XML; the rest of the modelling system (Fig. 2) could remain intact.

For most DSLs, the application domain was applied ecology rather than basic ecology, often in a spatial context and with an interface to GIS (Table 1). There we no DSLs for individual-based or agent-based modelling but Ginot et al. (2002) did outline how such a DSL could be constructed. The DSLs that we found were all external DSLs, i.e. ones that need an interpreter programmed for that DSL. Embedded or internal DSLs, which rely only on an existing programming language (see Fowler, 2011), thus represents a completely untested technique for ecological modelling. None of the DSLs used a functional language, like Scala (Subramaniam, 2008). Functional languages, whether in the context of DSLs or not, seem to represent yet another delayed take-up of computer science in ecological modelling.

The debate, whether DSLs are better than dedicated GUI development platforms, has its proponents and opponents, e.g. David et al. (2012) vs. Muetzelfeldt and Massheder (2003). It cannot be settled by argument but maybe with time. The two approaches might reflect trade-offs that depend on application domain (big vs. small models, teaching vs. research), in which case typical use patterns would emerge. But if is more a matter of personal preference then the debate will be never-ending. The dichotomy was evident from the beginning, when Robertson et al. (1989) argued that ecologists should have DSLs available formulated strictly in ecological semantics, and Durnota (1994) on the other hand envisaged that 'domain-specific user interfaces, which are built on top of a formal specification engine, can be constructed which would allow formally-naive, but domain knowledgeable users to specify their systems'. Fowler (2011) mentions the upcoming technology of a 'DSL workbench' which could maybe reconcile these opposing methodologies. Rahman et al. (2003) contrast DSLs with general programming languages, noting that 'it is much easier to achieve good runtime performance using commercially available compilers than by developing a domain specific language. Custom modelling languages often lack the flexibility of a commercial development tool that may limit their applicability to larger modelling applications'. However, this argument is faulty, as there is nothing preventing a DSL from relying on an efficient programming language or compiler. Furthermore, DSLs are limited in flexibility by purpose; they are highly flexible and expressive only in the domain that they address. Already, Wenzel (1992) argued that his focus on ecosystem modelling would limit the structural patterns in a fertile way only.

There has been several suggestions, but none implemented, that a DSL could be used not only to compose models, but also to validate the behaviour and relationships between model build blocks, in other words to expand the semantic expressiveness of the DSL. There have been attempts at defining wide-stretched semantic models and ontologies but these have not yet been consolidated in tools for common use. The ambition was expressed by Robertson et al. (1989), 'it is desirable that the range of permitted input expressions should be as wide as possible', and later by Villa and Costanza (2000), 'it is thus very important that tools do not constrain the researcher's thinking space within a specific view of natural complexity, but rather allow free space for thought by endorsing knowledge models which allow flexibility and reorganization'. However, while freedom of expression seems to set the researchers free, it also becomes a burden if taken too far, because an unconstrained modelling vocabulary also offers very little guidance. Compare with the observation of Fall and Fall (2001) that for models implemented in a general-purpose programming language (exemplifying an unconstrained medium), the underlying model becomes hidden in the details of the computer code, making it difficult to compare the conceptual and implemented models, and to modify the model.

The nature of model building blocks has also been a topic of debate. Some modellers are against object-oriented frameworks as a basis to construct model building blocks, but this opposition in many cases is based on a misconception of object-oriented design (OOD). Thus Bian (2000) stressed the difference between object-oriented and component-based design (CBD), emphasising the benefits of CBD due to its reliance on interfaces; however, the importance of interfaces is inherent to OOD (Gamma et al., 1995; Seemann, 2012). What does set Bian's (2000) model apart is, that it follows the COM and OpenGIS standards, allowing models to be composed of components distributed on Internet. The authors of Simile (Muetzelfeldt and Massheder, 2003) demonstrated a rather idiosyncratic viewpoint on OOD, when they stipulated that Simile building blocks are 'object-based' not 'object-oriented' . However, their objects are fully compliant with standard OOD.

The major divide in building blocks is between those that are independent modules, which can be executed in standard operating systems following standard communication protocols, and those that are objects, which depend on a dedicated execution system. Some modellers argue for the module-based approach (e.g., Rahman et al., 2004; Rizzoli et al., 2005), but which approach is better really depends on the application domain. For the integration of big models of diverse origin, the module-based approach is the better choice, even more so, if the models to be integrated span several application domains (for example, hydrology, agronomy and landscape planning; see Peckham et al., 2013). However, one must take in heed the warning of Gijsbers and Gregersen (2005): 'the most difficult task is assuring similar semantics of the data exchange between components'. For new models, simpler models and models addressing a limited domain, the stringency offered by a framework will aid modelling development. However, it is an art to make model building blocks generic in the relevant dimensions, while also keeping them simple enough to grasp easily by the user. In the words of Holzworth et al. (2010): 'How do we develop something that is reusable in different contexts and yet not over-engineer the solution?'

A major objective of DSLs is ease-of-use. This is reflected in the DSL's usage of concepts and an abstraction level aligned with the user's specific domain knowledge. It is an important design question, whether the DSL

should only give access to composition of models from existing building blocks, or whether it should provide facilities to create all new building blocks. To take the two latest DSLs as examples, the XML for FlexSem allows the user to write equations (Listing 12), which gives much freedom, even to change basic building block behaviour. In contrast, the XML for Universal Simulator only allows model composition (Listing 13); all behaviour is defined in the C++-programming of the building blocks. It seems that most DSLs were created with two usage scenarios in mind (Fig. 2): (i) model development, in which building blocks are created and the building block library is maintained, and (ii) model composition, in which existing building blocks are combined and configured for a specific application. As a DSL developer, one should think carefully about how the user would apply the DSL in both scenarios.

When it comes to the design of the DSL, how to make it complete for the domain, yet simple, Robertson et al. (1989) pointed out the trade-offs: 'Extending the range of input expressions [i.e., of DSL syntax] provides a basis for improving [the] dimensions but requires the complexity of other dimensions to be correspondingly increased'. To strike this balance we must know our users well; how much complexity is needed, and with which mind-set do the users manage this complexity. Ecological modelling is inherently complex, yet the DSL must be simple. The DSL purposefully defines a constrained medium for model construction. Again, in the words of Robertson et al. (1989): 'Thus […] expressive flexibility is available only to the designers of the base models [i.e., of the building blocks], not to the users of the system — whose only means of controlling model structure is by creating data flows between models [using the DSL]' . With such strong opposing forces, complexity vs. simplicity, an agile development process (Martin, 2006) seems necessary to develop a useful DSL for ecological modelling.

The popularity of DSLs in software engineering is only slowly spreading to the ecological modelling community. Modellers who have applied DSL methodology unknowingly will benefit from adjusting their home-brewed DSLs to the firm theoretical background and experience that already exists (see Fowler, 2011). XML-based DSLs in particular would be improved by a more domain-specific syntax. Modellers should embrace the simplicity that DSL-based models offer. Most of the few DSLs that we found for ecological modelling were at their base not constrained to ecology at all; they were generic modelling DSLs applied to ecological modelling. The future will show us what DSLs, developed more specifically for the ecological modelling domain, will look like.

Acknowledgements

NH received funding from EU Seventh Framework Programme under the grant agreement no. 265865-PURE. GFB was supported by a grant from the Graduate School of Science and Technology at Aarhus University (AUFF-F2012-FLS-3-14).

Appendix

```
VDFO: DIL
VCOM: ALGI, ALG2, BACT
VENV: PHOS
VSTA: S1, S2, SB, Q1, Q2,QB, TOPH
EGLO: 1
C - INDEPENDENT
PHOS: DIL
PHOS,ALG1,ALG2,BACT,QI,Q2,QB:DIL
TOPH: ALGI(0), ALG2(0), BACT(0), Q1(0), Q2(0), QB(0);
COMPARTMENT = ALG1
S1: Q1
Q1: S1(0), Q1
ALG1: ALG1, S1(0), DIL;
COMPARTMENT = ALG2
S2: Q2
Q2: S2(O),Q2
ALG2:ALG2, S2(0), DIL;
COMPARTMENT = BACT
SB: ALG1, ALG2, BACT
QB: SB(0), QB
BACT: BACT, SB(0), DIL;
P0
ITER, HS1, HS2, HSB, G1MA, G2MA, GBMA
G1M1, Q01, P
G2M1, Q02, P
G1M1,G2M1,EFEX
DIL=TABLE(TIME}
PHOS=P0*DIL

DO WHILE (I.LE.INT(ITER).AND.PHOS.GE.1.)
  V1 = G1MA/(1+HS1/PHOS)
  V2 = G2MA/(I+HS2/PHOS)
  VB = GBMA/(I+HSB/PHOS)
  Q1 = Q1+V1
  Q2 = Q2+V2
  QB = QB+VB
  PHOS = PHOS - (V1*ALG1 + V2*ALG2 + VB*BACT)
  I = I+1
END DO

TOPH = ALG1(0)*Q1(0) + ALG2(0)*Q2(0) + BACT(0)*QB(0)
S1 = G1M1*(1 - EXP(-P*(Q1/Q01-1)))
Q1 = Q1*EXP(S1(0))
ALG1 = ALG1*(1-DIL)*EXP(S1(0))
SB = EFEX*(ALG1*G1M1 + ALG2*G2M1)/BACT
```

Listing 1. A sample of MOSES code modelling competitive algal growth. After Wenzel, 1992.

```
#parameters
file = e:\oomp\database\agri.db;

#world
file = e:\oomp\weather\a93-96.wth;
SineVariable T(6,7);

#pools
Wheat wheat;
Aphid avenae, dirhodum, padi;
Parasitoid aphidius;

#links
(avenae dirhodum padi) consumes (wheat) at (M 1)

(aphidius.adult.egglaying) infects
   ((avenae dirhodum padi)(nymph)(apterous alate)(uninfected)) at (N 0.5)

((c7)(larva adult)) consumes (avenae dirhodum padi) at (M 0.175)

(c7.larva) consumes (c7.larva) at (M 0.09)
```

Listing 2. Part of a script specifying a tri-trophic system for the OOMP framework. After Holst et al., 1997.

```
Module PREDATORS_module {
   input Variable DEER_DENSITY {
   }
   Variable KILLS_PER_PREDATOR {
      update Command u0 { Value = Graph0(DEER_DENSITY); }
   }
   flux Variable MIGRATION_NORTH {
      update Command u1 { Value =
PREDATOR_POPULATION*PREDATOR_MIGRATION_RATE*SL::RANDOM(0.1, 0.4); }
   }
   flux Variable MIGRATION_SOUTH {
      update Command u2 { Value =
PREDATOR_POPULATION*PREDATOR_MIGRATION_RATE*SL::RANDOM(0.1, 0.4); }
   }
   flux Variable MIGRATION_EAST {
      update Command u3 { Value =
PREDATOR_POPULATION*PREDATOR_MIGRATION_RATE*SL::RANDOM(0.1, 0.4); }
   }
   flux Variable MIGRATION_WEST {
      update Command u4 { Value =
PREDATOR_POPULATION*PREDATOR_MIGRATION_RATE*SL::RANDOM(0.1, 0.4); }
   }
   flux Variable MIGRATION_IN {
      update Command u5 {
         Value = MIGRATION_NORTH@S + MIGRATION_SOUTH@N + MIGRATION_WEST@E +
MIGRATION_EAST@W;
      }
   }
   flux Variable PREDATOR_BIRTHS {
      update Command u6 { Value = PREDATOR_POPULATION*PREDATOR_NATALITY; }
   }
   flux Variable PREDATOR_DEATHS {
      update Command u7 { Value = PREDATOR_POPULATION*PREDATOR_MORTALITY; }
   }
   Variable PREDATOR_MIGRATION_RATE {
      update Command u8 { Value = Graph1(KILLS_PER_PREDATOR); }
   }
   Variable PREDATOR_MORTALITY {
      update Command u9 { Value = Graph2(KILLS_PER_PREDATOR); }
   }
   Variable PREDATOR_NATALITY {
      update Command u10 { Value = Graph3(DEER_DENSITY); }
   }
   state Variable PREDATOR_POPULATION {
      init Command i11 { Value = 3000; }
      integrate Command i12 {
         Method = Euler;           Clamp = True;
         Value = PREDATOR_BIRTHS + MIGRATION_IN - PREDATOR_DEATHS -
      MIGRATION_NORTH - MIGRATION_SOUTH - MIGRATION_EAST - MIGRATION_WEST;
      }
   }
   LUT Graph1 { listData = ( (0,1), (0.4, 0.795), ...) }
   LUT Graph2 { listData = ( (0,1), (0.15, 0.8), ...) }
   LUT Graph3 { listData = ( (0,0.05), (10, 0.05), ...) }
}
```

Listing 3. Part of an SML script defining predator-prey dynamics. After Maxwell and Costanza, 1997.

```
function Main() {
  Runoff = Coeff*max(0.0, Rainfall - PET);
}
```

Listing 4. A MickL programme for a run-off model which can be read by the Open Modelling Engine (OME). From Rahman et al., 2004.

```
<MODULE NAME="L-V">
  <STOCK NAME="PREY">
    <INFLOW NAME="BIRTH">
      r_prey*PREY
    </INFLOW>
    <OUTFLOW NAME="PREDATION">
      pred_eff*PREY*PREDATOR
    </OUTFLOW>
  </STOCK>

  <STOCK NAME="PREDATOR">
    <INFLOW NAME="PREDATION">
      pred_eff*pred_conv*PREY*PREDATOR
    </INFLOW>
    <OUTFLOW NAME="DEATH">
      d_predator*PREDATOR
    </OUTFLOW>
  </STOCK>
</MODULE>
```

Listing 5. Part of an IMT script for Lotka–Volterra dynamics. After Villa, 2001.

```
File GameOfLife.sel:
Seles Model
Model Size: 200, 200
Time Units: na Step 1
Landscape Events:
  GameOfLife.lse
Variable-Output View Maps:
  CellState
  PrevCellState
Output Model Bounds:
  CellState: 1
  PrevCellState: 1
Global Variables:
  pInitial = 0.05
Output Frequency: 1

File GameOfLife.LSE:
LSEVENT: GameOfLife
DEFINITIONS
  LAYER: CellState, PrevCellState
  GLOBAL VARIABLE: pInitial
ENDDEF
INITIALSTATE
  INITIALSTATE = 1
  CellState = IF UNIFORM(0,1) < pInitial THEN 1 ELSE 0
ENDIS
RETURNTIME
  RETURNTIME = 1
  PrevCellState = CellState
ENDRT
TRANSITIONS
  TRANSITIONS = TRUE
  numNeighbs = 0
  OVER REGION CENTRED(1,1.5, EUCLIDEAN)
    DECISION PrevCellState EQ 1
    numNeighbs = numNeighbs + 1
  ENDFN
  CellState = IF (CellState EQ 0) AND (numNeighbs EQ 3) THEN 1
    ELSE IF (CellState EQ 1) AND (2 <= numNeighbs <= 3) THEN 1
    ELSE 0
ENDTR
```

Listing 6. A SELES script for Conway's Game of Life. From SELES, 2014.

```
<farm>
  <waterSupply>
    <waterTake>
      <id>A</id>
      <quantity>28000</quantity>
      <rate>100</rate>
    </waterTake>
    <waterTake>
      <id>B</id>
      <quantity>30000</quantity>
      <rate>100</rate>
    </waterTake>
  </waterSupply>
  <irrigation>
    <irrigator>
      <id>A</id>
      <flowRate>40</id>
    </irrigator>
    <irrigator>
      <id>B</id>
      <flowRate>39</id>
    </irrigator>
  </irrigation>
</farm>
```

Listing 7. A simplified XML script for the FarmSim model.
From Good, 2005.

```
<component class="org.unijena.refET.calcVPD" name="calcVPD">
  <jamsvar name="tmean" provider="TmeanDataReader" value="tmean"/>
  <jamsvar name="rhum" provider="RhumDataReader" value="rhum"/>
  <jamsvar name="vpd" value="vpd"/>
</component>
```

Listing 8. An XML script declaring a component for the JAMS framework.
From Kralisch and Krause, 2006.

```
<simulation name="A">
  <component name="B">
    <executable name="model_b.dll"/>
    <initdata>
    :
    </initdata>
  </component>

  <system name="C">
    <executable name="model_c.dll"/>
    <initdata>
    :
    </initdata>

    <component name="D">
      <executable name="model_d.dll"/>
      <initdata>
      :
      </initdata>
    </component>
    <component name="E">
      <executable name="model_e.dll"/>
      <initdata>
      :
      </initdata>
    </component>
  </system>
</simulation>
```

Listing 9. An XML configuration file for CMP. Left-out code denoted by ':'.
From Moore et al., 2007.

```
relationdef Grazing(grass, herbivores) {
  number value=grass.getSurfaceArea();
  herbivores.addBiomassIngested(value*conversioncoeff);
}
```

Listing 10. A 'grazing' relation between entities, defined in Ocelet.
From Degenne et al., 2009.

```
sim(name:"TW") {
  build(targets:"all")
  outputstrategy(dir: "$oms_prj/output", scheme:SIMPLE)

  // define model
  model(iter:"climate.moreData") {
    components {
      climate 'tw.Climate'
      daylen 'tw.Daylen'
      et 'tw.HamonET'
      out 'tw.Output'
      runoff 'tw.Runoff'
      snow 'tw.Snow'
      soil 'tw.SoilMoisture'
    }
    connect { // 'source' 'target'
      // climate
      'climate.temp' 'soil.temp'
      'climate.temp' 'et.temp'
      'climate.temp' 'snow.temp'
      'climate.precip' 'soil.precip'
      'climate.precip' 'snow.precip'
      'climate.time' 'daylen.time'
      'climate.time' 'et.time'
      'climate.time' 'out.time'
      // daylen
      'daylen.daylen' 'et.daylen'
      'daylen.daylen' 'out.daylen'
      // soil
      'soil.surfaceRunoff' 'out.surfaceRunoff'
      'soil.surfaceRunoff' 'runoff.surfaceRunoff'
      'soil.soilMoistStor' 'out.soilMoistStor'
      'soil.actET' 'out.actET'
      // PET
      'et.potET' 'soil.potET'
      'et.potET' 'snow.potET'
      'et.potET' 'out.potET'
      // Snow
      'snow.snowStorage' 'out.snowStorage'
      'snow.snowMelt' 'runoff.snowMelt'
      // runoff
      'runoff.runoff' 'out.runoff'
    }
    parameter { // 'comp.field' value
      'climate.climateInput' "$oms_prj/data/climate.csv"
      'out.outFile' "$oms_prj/output/TW/out/output.csv"
      'runoff.runoffFactor' 0.5
      'daylen.latitude' 35.0
      'soil.soilMoistStorCap' 200.0
    }
  }
}
```

Listing 11. An example of one of the OMS3 dialects.
From David et al., 2012.

```
<Input>
  <Filename>..\..\data\Horsens_met2004.txt</Filename>
  <Format>TEXT0D</Format>
  <Item>
    <ItemNumber>2</ItemNumber>
    <Symbol>glorad</Symbol>
  </Item>
  <Item>
    <ItemNumber>4</ItemNumber>
    <Symbol>precip</Symbol>
  </Item>
</Input>

<EcoLight>
  <IncommingRadiation>glorad</IncommingRadiation>
  <Salinity>tracelightvar</Salinity>
  <DetritusCarbon>X</DetritusCarbon>
  <PlanktonNCRatio>SFromFile</PlanktonNCRatio>
  <XNratio>0.037</XNratio>
  <Kw>0.593</Kw>
  <EpsilonS>9.3</EpsilonS>   <!-- 4.3 Distance/tracer led 7 -->
  <EpsilonC>0.015</EpsilonC>   <!-- 0.015 Chl led -->
  <EpsilonX>1</EpsilonX>
  <R1>0.55</R1>
  <AddkDasVariable>true</AddkDasVariable> <!--kD-->
 </EcoLight>

<Equation>
  <Definition>X=qXN*N</Definition>
  <Description>Chlorophyll</Description>
</Equation>
```

Listing 12. Part of an XML script file for Flexsem.
After Larsen et al., 2013.

```
<model name="colony">
  <model name="egg" type="Vespa::Egg">
    <parameter name="newEggs" ref="../queen[eggsLaid"/>
    <parameter name="newFertilisedEggs" ref=".[newEggs]"/>

    <model name="mass" type="UniSim::Stage">
      <parameter name="duration" value="2"/>
      <parameter name="k" value="30"/>
    </model>

    <model name="number" type="UniSim::Stage">
      <parameter name="duration" ref="../mass[duration]"/>
      <parameter name="k" ref="../mass[k]"/>
    </model>
  </model>

  <model name="queen" type="Vespa::Queen">
  :
  </model>
</model>
```

Listing 13. Part of an XML script for Universal Simulator. Left-out code denoted by ':'.
After Holst, 2014.

References

Adam, M., Corbeels, M., Leffelaar, P.A., van Keulen, H., Wery, J., Ewert, F., 2012. Building crop models within different crop modelling frameworks. Agric. Syst. 113, 57–63.

Argent, R.M., 2004. An overview of model integration for environmental applications — components, frameworks and semantics. Environ. Model Softw. 19, 219–234.

Argent, R.M., 2007. E2 — past, present and future. In: Oxley, L., Kulasiri, D. (Eds.), Modsim 2007: International Congress on Modelling and, Simulation, pp. 860–866.

Argent, R.M., Voinov, A., Maxwell, T., Cuddy, S.M., Rahman, J.M., Seaton, S., Vertessy, R.A., Braddock, R.D., 2006. Comparing modelling frameworks — a workshop approach. Environ. Model Softw. 21, 895–910.

Baveco, J.M., Smeulders, A.M.W., 1994. Objects for simulation — smalltalk and ecology. Simulation 62, 42–56.

Bian, L., 2000. Component modeling for the spatial representation of wildlife movements. J. Environ. Manag. 59, 235–245.

CCA, 2014. Common Component Architecture. www.cca-forum.org.

Chabrier, P., Garcia, F., Martin-Clouaire, R., Quesnel, G., Raynal, H., 2007. Toward a simulation modeling platform fro studying cropping systems management: the record project. In: Oxley, L., Kulasiri, D. (Eds.), Modsim 2007: International Congress on Modelling and Simulation, pp. 839–845.

Constanza, R., Voinov, A., 2003. Modeling ecological and economic systems with STELLA: part III. Ecol. Model. 143, 1–7.

David, O., Lloyd, W., Ascough II, J.C., Green, T.R., Olson, K., Leavesley, G.H., Carlson, J., 2012. Domain specific languages for modeling and simulation: use case OMS3. In: Seppelt, R., Voinov, A.A., Lange, S., Bankamp, D. (Eds.), International Congress on Environmental Modelling and Software, 1–5 July 2012 (Leipzig, Germany).

Dearle, F., 2010. Groovy for domain-specific languages. Packt Publishing, Birmingham, UK.

Degenne, P., Lo Seen, D., Parigot, D., Forax, R., Tran, A., Lahcen, A.A., Cure, O., Jeansoulin, R., 2009. Design of a domain specific language for modelling processes in landscapes. Ecol. Model. 220, 3527–3535.

Degenne, P., Ait Lahcen, A., Curé, O., Forax, R., Parigot, D., Lo Seen, D., 2010. Modelling the environment using graphs with behaviour: do you speak Ocelet? In: Swayne, D.A., Yang, W., Voinov, A.A., Filatova, T. (Eds.), International Congress on Environmental Modelling and Software, 5–8 July 2010, Ottawa, Canada (8 pp)

Derry, J.F., 1998. Modelling ecological interaction despite object-oriented modularity. Ecol. Model. 107, 145–158.

Durnota, B., 1994. Defining relationships in ecology using object-oriented formal specifications. Math. Comput. Model. 20, 83–96.

Eclipse, 2014. www.eclipse.org.

Fall, A., Fall, J., 2001. A domain-specific language for models of landscape dynamics. Ecol. Model. 141, 1–18.

Fowler, M., 2011. Domain-specific Languages. Addison-Wesley, Upper Saddle River, NJ.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns. Elements of Reusable Object-oriented Software. Addison-Wesley Publishing Company, Reading, Massachusetts.

Gaucherel, C., Giboire, N., Viaud, V., Houet, T., Baudry, J., Burel, F., 2006. A domain-specific language for patchy landscape modelling: the Brittany agricultural mosaic as a case study. Ecol. Model. 194, 233–243.
Gijsbers, P.J.A., Gregersen, J.B., 2005. OpenMI: a glue for model integration. In: Zerger, A., Argent, R.M. (Eds.), Modsim 2005. International Congress on Modelling and Simulation, 12–15 December 2005, Nedlands, Australia, pp. 648–654.
Ginot, V., Le Page, C., Souissi, S., 2002. A multi-agents architecture to enhance end-user individual based modelling. Ecol. Model. 157, 23–41.
Good, J., 2005. The benefits and practicalities of using extensible markup language (XML) for the interfacing and control of object-oriented simulations. In: Zerger, A., Argent, R.M. (Eds.), Modsim 2005. International Congress on Modelling and Simulation, 12–15 December 2005, Nedlands, Australia, pp. 655–661.
Harvey, H., 2005. Languages and metamodels for modelling frameworks. In: Zerger, A., Argent, R.M. (Eds.), Modsim 2005. International Congress on Modelling and Simulation, 12–15 December 2005, Nedlands, Australia, pp. 669–675.
He, H.S., Larsen, D.R., Mladenoff, D.J., 2002. Exploring component-based approaches in forest landscape modeling. Environ. Model Softw. 17, 519–529.
Holst, N., 2010. WeedML: a tool for collaborative weed demographic modeling. Weed Sci. 58, 497–502.
Holst, N., 2013. A universal simulator for ecological models. Ecol. Inform. 13, 70–76.
Holst, N., 2014. Universal Simulater Explained. www.ecolmod.org.
Holst, N., Axelsen, J.A., Olesen, J.E., Ruggle, P., 1997. Object-oriented implementation of the metabolic pool model. Ecol. Model. 104, 175–187.
Holzworth, D.P., Huth, N.I., de Voil, P.G., 2010. Simplifying environmental model reuse. Environ. Model Softw. 25, 269–275.
Hucka, M., Finney, A., Sauro, H.M., Bolouri, H., Doyle, J.C., Kitano, H., and the rest of the, S.F., Arkin, A.P., Bornstein, B.J., Bray, D., Cornish-Bowden, A., Cuellar, A.A., Dronov, S., Gilles, E.D., Ginkel, M., Gor, V., Goryanin, I.I., Hedley, W.J., Hodgman, T.C., Hofmeyr, J.H., Hunter, P.J., Juty, N.S., Kasberger, J.L., Kremling, A., Kummer, U., Le Novere, N., Loew, L.M., Lucio, D., Mendes, P., Minch, E., Mjolsness, E.D., Nakayama, Y., Nelson, M.R., Nielsen, P.F., Sakurada, T., Schaff, J.C., Shapiro, B.E., Shimizu, T.S., Spence, H.D., Stelling, J., Takahashi, K., Tomita, M., Wagner, J., Wang, J., 2003. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. Bioinformatics 19, 524-531.
Jones, J.W., Keating, B.A., Porter, C.H., 2001. Approaches to modular model development. Agric. Syst. 70, 421–443.
Kralisch, S., Krause, P., 2006. JAMS — a framework for natural resource model development and application, iEMSs Third Biannual Meeting, 9–13 July 2006. Burlington, Vermont, USA (6 pp).
Larkin, T.S., Carruthers, R.I., Soper, R.S., 1988. Simulation and object-oriented programming: the development of SERB. SIMULATION 51, 93–100.
Larsen, J., Mohn, C., Timmermann, K., 2013. A novel model approach to bridge the gap between box models and classic 3D models in estuarine systems. Ecol. Model. 266, 19–29.
Liu, J.G., Ashton, P.S., 1995. Individual-based simulation-models for forest succession and management. For. Ecol. Manag. 73, 157–175.
Martin, R.C., 2006. Agile software developmen. Principles, Patterns and Practices. Prentice Hall, Upper Saddle River, New Jersey.
Martin, R.C., 2009. Clean code. A Handbook of Agile Software Craftsmanship. Prentice-Hall, Upper Saddle River, New Jersey.
Maxwell, T., Costanza, R., 1997. A language for modular spatio-temporal simulation. Ecol. Model. 103, 105–113.
Merali, Z., 2010. Why scientific programming does not compute. Nature 467, 775–777.
Moore, A.D., Holzworth, D.P., Herrmann, N.I., Huth, N.I., Robertson, M.J., 2007. The common modelling protocol: a hierarchical framework for simulation of agricultural and environmental systems. Agric. Syst. 95, 37–48.
Muetzelfeldt, R., Massheder, J., 2003. The Simile visual modelling environment. Eur. J. Agron. 18, 345–358.
Papajorgji, P., Beck, H.W., Braga, J.L., 2004. An architecture for developing service-oriented and component-based environmental models. Ecol. Model. 179, 61–76.
Parr, T., 2009. Language implementation patterns. Create Your Own Domain-Specific and General Programming Languages. The Pragmatic Bookshelf, Dallas, Texas.
Patrick Smith, F., Holzworth, D., Robertson, M.J., 2005. Linking icon-based models to code-based models: a case study with the agricultural production systems simulator. Agric. Syst. 83, 135–151.
Peckham, S.D., Hutton, E.W.H., Norris, B., 2013. A component-based approach to integrated modeling in the geosciences: the design of CSDMS. Comput. Geosci. 53, 3–12.
Peng, C.H., 2000. Growth and yield models for uneven-aged stands: past, present and future. For. Ecol. Manag. 132, 259–279.
R Development Core Team, 2014. R: a language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. www.r-project.org.
Rahman, J.M., Seaton, S.P., Perraud, J.M., Hotham, H., Verrelli, D.I., Coleman, J.R., Mssanzi, 2003. It's TIME for a new environmental modelling framework. MODSIM 2004. International Congress on Modelling and Simulation. Townsville, Australia, pp. 1727–1732.
Rahman, J.M., Seaton, S.P., Coddy, S.M., 2004. Making frameworks more useable: using model introspection and metadata to develop model processing tools. Environ. Model Softw. 19, 275-184.
Reed, M., Cuddy, S.M., Rizzoli, A., 1999. A framework for modelling multiple resource management issues—an open modelling approach. Environ. Model Softw. 14, 503–509.
Reynolds, J.F., Acock, B., 1997. Modularity and genericness in plant and ecosystem models. Ecol. Model. 94, 7–16.
Rizzoli, A., Donatelli, M., Athanasiadis, I., Villa, F., Muetzelfeldt, R., Huber, D., 2005. Semantic links in integrated modelling frameworks. In: Zerger, A., Argent, R.M. (Eds.), MODSIM 2005. International Congress on Modelling and Simulation, 12–15 December 2005, Nedlands, Australia, pp. 704–710.
Robertson, D., Bundy, A., Uschold, M., Muetzelfeldt, R., 1989. The ECO program construction system: ways of increasing its representational power and their effects on the user interface. Int. J. Man Mach. Stud. 31, 1–26.
Seemann, M., 2012. Dependency Injection in.NET. Manning Publ. Co, Shelter Island, NY, USA.
SELES, 2014. www.seles.info/index.php/Game_of_Life.
Sequeira, R.A., Olson, R.L., McKinion, J.M., 1997. Implementing generic, object-oriented models in biology. Ecol. Model. 94, 17–31.
Silvert, W., 1993. Object-oriented ecosystem modeling. Ecol. Model. 68, 91–118.
Subramaniam, V., 2008. Programming Scala. The Pragmatic Bookshelf, Dallas, Texas.
Tisue, S., Wilensky, U., 2004. NetLogo: design and implementation of a multi-agent modeling environment. Proceedings of Agent 2004, Chicago, October 2004 (17 pp).
van Deursen, A., Klint, P., Visser, J., 2000. Domain-specific languages: an annotated bibliography. ACM SIGPLAN Not. 35, 26–36.
van Evert, F., Holzworth, D., Muetzelfeldt, R., Rizzoli, A., Villa, F., 2005. Convergence in integrated modeling frameworks. In: Zerger, A., Argent, R.M. (Eds.), Modsim 2005. International Congress on Modelling and Simulation, 12–15 December 2005, Nedlands, Australia, pp. 745–750.
Villa, F., 2001. Integrating modelling architecture: a declarative framework for multi-paradigm, multi-scale ecological modelling. Ecol. Model. 137, 23–43.
Villa, F., Costanza, R., 2000. Design of multi-paradigm integrating modelling tools for ecological research. Environ. Model Softw. 15, 169–177.
Wenzel, F., 1992. Semantics and syntax elements and a unique calculus for modelling of complex ecological systems. Ecol. Model. 63, 113–131.
Wickham, H., 2015. Advanced R. CRC Press, Boca Raton, Florida (URL:adv-r.had.co.nz/dsl.html.).
Zeigler, B.P., 1987. Hierarchical, modular discrete-event modelling in an object-oriented environment. Simulation 49, 219–230.