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

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/© 2015 Elsevier B.V. All rights reserved.

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