Europ. J. Agronomy 18 (2003) 345–358

European Journal of Agronomy

www.elsevier.com/locate/eja

The Simile visual modelling environment

Robert Muetzelfeldt *, Jon Massheder

The University of Edinburgh, Institute of Ecology and Resource Management, Darwin Building, King's Buildings, Mayfield Road, Edinburgh EH9 3JU, Scotland, UK

Abstract

Simile is a visual modelling environment that has been developed to overcome the problems involved in implementing agro-ecological simulation models using conventional programming languages: problems such as the effort and skill needed to program the models, the lack of transparency in models implemented as programs, and the lack of re-useability of models and submodels. It combines the familiar System Dynamics (compartment-flow) paradigm with an object-based paradigm, allowing many forms of disaggregation to be handled, as well as spatial modelling and individual-based modelling. Its visual modelling interface makes it accessible to non-programmers, at the same time allowing models to be largely self-documenting. Models can be run very efficiently as compiled C++ programs, and users can develop new visualisation tools for displaying model results. Simile has been used in international research programmes, including the modelling of Mediterranean vegetation dynamics and modelling the interaction between households and land at the forest margin in developing countries. Simile has been developed in a spirit of open standards for model sharing. Models are saved as a text file in a structured format, with a view to enable model sharing with other modelling environments and to encourage others to develop additional tools for working with models.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Modelling; Modelling environment; Simile; Declarative modelling

1. Introduction

For a number of years, there has been considerable unease with the traditional method of implementing agro-ecological models, based on writing computer programs to solve the model equations (Reynolds and Acock, 1997). Generally, this has been expressed in terms of the desirability of a modular modelling approach, in order to avoid duplication of effort by allowing re-use of standard modules for (e.g.) crop growth or soil water dynamics. The issues run deeper than this, however, and relate to the role of modelling as a scientific activity, calling for complete transparency, reproducibility and verification of models as components of scientific argumentation.

Various solutions have been proposed:

(1) Improved software engineering practices for implementing models in conventional programming languages. This has in general involved the development of standards for the programming interface of models and the submodels they contain, in order to improve modularity and to allow

* Corresponding author. Tel.: +44-131-650-5408; fax: +44-131-662-0478
E-mail addresses: r.muetzelfeldt@ed.ac.uk (R. Muetzelfeldt), j.massheder@ed.ac.uk (J. Massheder).

for the development of generic control routines and input/output routines. The rationale here is that, if models are going to be implemented as computer programs, this might as well be done in a principled manner, with clean distinctions between the module(s) containing the model itself and those concerned with running the model. One example of such an approach is FSE, the Fortran Simulation Environment (van Kraalingen, 1995), which is being used as the basis for the re-implementation of the DSSAT family of crop models (Porter et al., 1999). APSIM (McCown et al., 1996), a crop modelling environment, exemplifies the development of a fixed framework for modular modelling.

(2) Simulation languages for models based on sets of differential-algebraic equations have been around since the 1970s. In Dynamo (Richardson and Pugh, 1981), the equations are expressed in terms of System Dynamics, while in continuous systems modelling program (CSMP) and advanced computer simulation language ((ACSL): http://www.acslsim.com/) the equations relate directly to differential equations: the differences are small and essentially cosmetic. CSMP was promoted extensively by crop modellers in Wageningen in the 1970s (de Wit and Goudriaan, 1974), but is now essentially dead, while ACSL has been used more recently for some important grass and crop models (Johnson and Thornley, 1985).

(3) Visual modelling environments, based on the System Dynamics paradigm for continuous systems modelling, have dramatically improved the accessibility of modelling tools to students and researchers with few programming or mathematical skills. The main contenders (Stella, ModelMaker, Vensim and Powersim: see Reference section for URLs) offer similar features, and have been used for serious research modelling: indeed, three special issues of the journal Ecological Modelling have been devoted to Stella models alone (Costanza et al., 1998; Costanza and Gottlieb, 1998; Costanza and Voinov, 2001).

(4) Object-oriented (OO) and component-based approaches are increasingly popular. An OO approach emphasises the correspondence between objects in the real world and 'objects' in the software engineering sense, and allows for the inheritance of attributes and behaviours from more general to more specific classes. It tends to be used within a particular modelling project, and is seen as a valuable way of increasing the modularity of model design: indeed, a special issue of the journal Ecological Modelling on 'Modularity in Plant Models' concentrated exclusively on OO methodology (Acock and Reynolds, 1997; Sequeira et al., 1997; Acock and Reddy, 1997; Lemmon and Chuk, 1997; Chen and Reynolds, 1997; Timlin and Pachepsky, 1997). Component-based methods, on the other hand, tend to be developed to allow the integration and interoperability of modules, often using existing code, implemented in different languages and on various platforms. Potter et al. (2000) evaluated the use of Microsoft's Distributed Component Object Model in forest ecosystem modelling, while van Evert and Bolte (this issue) describe MODCOM, a system which aims to facilitate the assembly of simulation models from previously and independently developed component models. However, it is notable that the community is still very fragmented: no common standards have emerged, and incompatible approaches are used by different groups.

Simile is our attempt at addressing the problems with modelling. It uses a visual modelling interface because we believe that provides the best way for building, analysing and communicating models. But it aims to overcome the deficiencies of existing visual modelling environments by providing far greater expressiveness, including the ability to handle disaggregation, spatial modelling, and dynamically-varying populations of objects. It provides for efficient simulation of complex models as compiled C++ programs. And it is based on the view that modelling is a design activity, and that an essential requirement for collaborative modelling is the development of a common, standard language capable of representing models as designs. More information on Simile can be found at http://www.ierm.ed.ac.uk/simile.

2. Main features

Simile is derived from Agroforestry Modelling Environment (AME), the AME (Muetzelfeldt and

Taylor, 1997a,b). While Simile is a generic modelling environment, the original requirement to handle process-based interactions in complex tree-crop systems has led to it having a number of features not usually found together. Simile extends AME in a number of ways, including the ability to have external data files and a new, single-window user interface for running simulations.

2.1. *Visual modelling*

In common with other visual modelling environments, Simile supports a two-phase approach to model construction. The first phase involves the drawing of diagrams that show the main features of the model. The second phase involves fleshing-out the model-diagram elements with quantitative information: values and equations.

2.2. *System Dynamics*

Simile allows models to be formulated in System Dynamics terms: that is, as compartments (stocks, levels) whose values are governed by flows in and flows out. This can be considered as a visual language for representing differential-equation models, with a compartment representing a state variable, and the rate-of-change being the net sum of inflows minus outflows. This commitment to System Dynamics means that we restrict the modelling to continuous-time/discrete-time systems (differential/difference equations): Simile does not support discrete-event modelling. However, the vast majority of agro-ecological models are based on continuous or discrete time, so this is hardly a restriction.

2.3. *Disaggregation*

Simile allows the modeller to express many forms of disaggregation: e.g. age/size/sex/species classes. This is done by defining how one class behaves, then specifying that there are many such classes.

2.4. *Object-based modelling*

Simile allows a population of objects, such as animals or trees, to be modelled. As with disaggregation, you define how one member behaves, then specify that there are many such members. In this case, the model designer provides rules for specifying when new members of the population are created, and for killing off existing members. Individual members of the population can interact with others.

2.5. *Spatial modelling*

Spatial modelling, in Simile, is simply a special form of disaggregation. One spatial unit (grid square, hexagon, polygon...) is modelled, then many such units are specified. Each spatial unit can be given spatial attributes (area, location), and the proximity of one unit to another can be represented.

2.6. *Modular modelling*

Simile allows any Simile model to be inserted as a submodel into another Simile model. Having done this, the modeller can then manually make the links between variables in the two components (in the case where the submodel was not designed to plug into the main model); or links can be made automatically, giving a ‘plug-and-play’ capability. Conversely, any submodel can be extracted and run as a stand-alone model (‘unplug-and-play’), greatly facilitating the testing of submodels of a complex model.

2.7. *Fast simulation*

Models can be run as compiled C++ programs. In many cases, these will run at speeds similar to a hand-coded program, enabling Simile to cope with complex models (100s equations; 1000s object instances).

2.8. *Customisable output displays and input tools*

Simile users can design and implement their own input/output procedures, independently of the

Simile developers. In particular, one can develop displays for model output that are specific to one's own particular requirements. Once developed, these can be shared with others in the research community.

2.9. Declarative representation of model structure

Most agro-ecological models are implemented *procedurally* —as a set of instructions for simulating the behaviour of the model, programmed in a conventional programming language such as Fortran or C. Simile, in common with simulation languages and other visual modelling software, represents models *declaratively* —as a set of statements defining the structure of the model. These statements are stored in an open, structured format in a text file (unlike some modelling packages, which use a proprietary binary format). This approach allows for other groups to develop software for processing Simile models. For example, one group may develop a new way of reporting on model structure, while another may wish to undertake automatic comparison of the structure of two similar models. It also opens the way for the sharing of models between different modelling environments.

3. The modelling language

Within the domain of continuous-time/discrete-time modelling, Simile aims to provide a 'one-stop shop': to enable modellers to build whatever models they want within a single environment. This is highly ambitious: there are many different types of models: statistical, process-based, spatial, Leslie matrix, Markov chain, cellular automaton, etc. How can a single language cope with all these types?

Clearly, we cannot have constructs in the language specific to each type of model: the language would have a large number of constructs, and be very difficult to learn. Instead, we need to come up with a set of low-level modelling constructs (primitives) that enable the various types of models to be constructed, without necessarily using the terms or concepts associated with each model type. In a sense, a programming language (such as C) does this already—but its constructs are at too low a level, too far removed from the concepts used by the modeller, hence requiring large programs and specialist skills. The challenge for the designer of a modelling language is to avoid a proliferation of symbols in the language, while at the same time retaining intuitiveness and expressiveness.

Our solution to this problem is based on a formal language that enables models to be specified in terms of 12 basic elements. Although in principle one can construct a model by creating a text file in this language, this would be a difficult and error-prone process, so we have produced a graphical user interface which enables a model to be created diagrammatically, using 11 icons (Fig. 1). These symbols fall naturally into two classes: those concerned with standard System Dynamics concepts, and those based on submodels and a notion of 'object'. These icons have a direct correspondence with the elements in the underlying formal language, with the exception that the 'function' element does not have an icon.

3.1. System Dynamic components

System Dynamics notation (Forrester, 1961) is an intuitive and widely-used way of describing continuously-varying systems. It is a particularly appropriate language for describing ecological systems, since it combines concepts of amount, flow and influence: many ecological researchers use such notation to describe the system they are investigating even if they have no experience in modelling. Most of the visual modelling environments used in ecological and agronomic research and education are based on System Dynamics

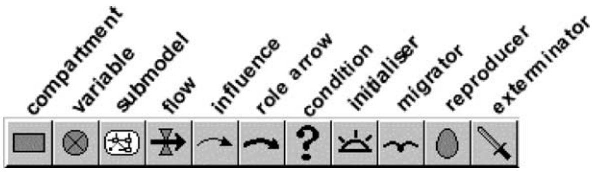

Fig. 1. The eleven symbols that constitute Simile's diagramming language. See text for an explanation of the role of each symbol.

notation. Therefore, we made a commitment early on to include System Dynamics notation in Simile's visual modelling language.

System Dynamics, as realised in Simile, is based on four symbols:

(1) The *compartment* (also known as stock, level) can be thought of as representing the amount of some substance (such as money, water, carbon or nitrogen). More generally, the compartment is a visual representation of a mathematical state variable, and thus can be used to represent quantities whose behaviour is governed by a differential equation but which do not really represent 'amount of substance': e.g. height of a tree or position of a moving object.

(2) The *flow arrow* represents a process that contributes to the rate of change of a compartment. A compartment may have multiple inflows and outflows: the net rate of change of the compartment is the sum of the inflows minus outflows.

(3) A *variable* represents some other quantity in the system under consideration. A variable in Simile can represent a parameter, an intermediate variable, an exogenous variable or an output variable, depending on its influence links with other variables and the expression used to calculate its value.

(4) The *influence arrow* shows visually which quantities are used to calculate which other quantities.

3.2. The submodel and associated symbols

The submodel construct is the key to Simile's ability to handle a wide variety of model-design requirements.

Essentially, the submodel is a container for some collection of model elements, including System Dynamics elements and other submodels. At the simplest level, it can be used to divide a complex model visually into different sections-rather like the 'sector' in Stella. Used in this way, it has no implications for the mathematical structure of the model. The submodel can also be used for modular modelling, since a submodel can be saved to file and loaded from file independently of the model it is in, rather like the 'co-model' in Powersim or the 'submodel' in ModelMaker.

However, the real power of the submodel comes when we specify that there are multiple instances of a submodel. This is roughly analogous to the notion of 'class' and 'object' in OO software engineering: the submodel represents the class, and the multiple instances represent multiple objects belonging to the class. In this section, we briefly summarise some uses of the submodel from a modeller's perspective, introducing the remaining model-design symbols in the process.

3.2.1. Disaggregation

Modellers frequently divide some component into a fixed number of elements. This is generically referred to as 'disaggregation'. For example:

- a population may be divided into a number of age, size, or sex classes;
- a vegetation component may be divided into the several species that make it up;
- soil, or a forest canopy, may be divided into a number of layers;
- an area may be divided into grid squares, polygons, or some other form of spatial unit.

The usual way of allowing this in visual modelling environments is to use an array structure for every disaggregated variable. In Simile, in contrast, one wraps the appropriate model elements in a submodel—then simply specifies a number of instances for this submodel as a submodel property. Visually, the submodel is drawn with a 'stack of cards' boundary. Each instance can be given specific attributes, e.g. x and y co-ordinate and soil type for spatial grid-squares.

3.2.2. Individual-based modelling

Individual-based modelling (Grimm, 1999) is generally seen as an alternative to System Dynamics modelling, and modelling environments have been developed that specifically support this approach (Lorek and Sonnenschein, 1999). Simile supports individual-based modelling, by allowing a population of individuals to be specified, while at the same time allowing the behaviour of each individual, or of other components of the system, to be expressed in terms of System Dynamics.

For example, a modeller might represent the vegetation in terms of compartments and flows, while the herbivores might be represented as individual animals, which are created, grow and die. In order to do this, a submodel is specified as being a population submodel (again, in its Properties box). Visually, the submodel now appears with a shadow line for the top- and left-edges, and another for the bottom- and right-edges. Model elements (Initialiser, Migrator, Reproducer and Eliminator (Fig. 1) can be added for specifying: the initial number of instances; the rules for the creation of new individuals in absolute numbers or for each existing individual; and the elimination of those already in the population.

3.2.3. Conditional existence of some part of the model

Modellers frequently need to be able to specify the conditional existence of some part of a model. For example:

- You may want to have several alternative ways of modelling some part of the system (e.g. a growth function), only one of which is active in any one run of the model. A flag determines which one is active.
- You may want to model a set of species using a single submodel, but with only some species present in any one run of the model.
- You may want to model a number of spatial patches, some of which contain one land use type, and others of which contain another. You need to include a submodel for each one within the multiple-instance patch submodel—but switch one or the other on in a particular patch.

All these situations can be handled in Simile using a conditional submodel. This is simply a normal submodel, but with a condition symbol added (Fig. 1). Visually, we can tell that it is a conditional submodel both by the presence of the condition symbol, and by a set of dots going down diagonally to the right from the submodel envelope. The condition contains a boolean expression: if this evaluates to 'true', then the submodel (or an instance of it) exists; if not, then it does not.

3.2.4. Using a submodel to specify an association between objects

Once our modelling language allows us to think in terms of multiple objects of a certain type, then it is frequently the case that we start to recognise relationships between objects. These relationships may be:

- between objects of the same type: one tree <u>shades</u> another; one grid square <u>is next to</u> another; one person <u>is married to</u> another; or
- between objects of one type and objects of another: one farmer <u>owns</u> a field; one field <u>is close</u> to a village.

Since Simile is a visual modelling language, and since such relationships are an important aspect of the design of a particular model, Simile provides visual elements to show diagrammatically such relationships between objects. Since the term 'relationship' is normally used in Ecological Modelling to refer to a relationship between *variables* (as opposed to *objects*), we use the term 'association' instead, in line with the terminology of the Unified Modelling Language (UML) (Stevens and Pooley, 2000).

An association can itself have properties. We can, for example, have a variable representing the actual distance between a field and a village: this is a property of neither the field nor the village, but of the association between them. In Simile, the submodel is the construct that is able to hold a number of quantities, so we use a submodel to represent an association. We show which objects have a role in the association by drawing a role arrow (Fig. 1) from the submodel representing the object to the association submodel, one for each of the two roles in the association.

The association submodel also has an important role to play in passing information between the objects that participate in the association: for example, information on the yield from each of a farmer's fields would be pass through the association submodel specifying which farmers own which fields. This can greatly improve the computational efficiency of complex models, since processing occurs only for the interactions where an association exists, not for all possible interactions between objects.

3.3. Data structures and the equation language

3.3.1. Data types

Simile currently supports three data types: real (floating point), integer, and booleans. In general, Simile will infer the type of a quantity from the expression used to produce it. During the model-design process, it continually checks for consistency. For example, if A influences B and the expression 2*A is entered for B, then Simile assumes that B is numeric. If A is subsequently given a boolean value, it flags that the expression for B is incorrect.

3.3.2. Data structures

Simile handles scalars, arrays (of multiple dimensions) and lists. Data structures can be nested to any depth. So you can have an array in which each element is itself an array, each element of which is a list of scalars.

In many cases, the data structure will be created automatically by Simile. For example, if you create a submodel with 10 instances, and take an influence from the submodel to a variable outside it, then the quantity exported from the submodel is automatically a 10-element array. If you do the same thing for a population model, then the exported quantity is a list (since the number of values is not pre-determined).

The user can also create data structures explicitly. For example, if you enter the expression [10,20,30,40,50] in the equation box for a variable, then its value is an array with 5 elements. Simile will automatically recognise this, and, as with data types, maintain consistency checking that the variable is used correctly in the equations for other quantities.

3.3.3. Equation language

Simile provides the standard mathematical operators and functions, as well as functions specific to Simile (e.g. for array processing and for simulation). Users can define their own functions, which are simply edited into a text file in the Simile directory, the only constraint being that the function is expressible in Simile's equation language. Users can thus share user-defined functions with each other.

Expressions can be conditional statements (if...then...elseif...then...else), which can be nested to any depth (i.e. any subexpression can itself be a conditional expression).

Simile provides a powerful array-processing language, so that the user does not have to resort to procedural programming constructs (e.g. loops) for processing arrays. For example, if [A] and [B] are both 5-element arrays, then [A]*[B] creates another 5-element array where each element consists of the corresponding elements of A and B multiplied together.

3.4. Putting it all together: an example model

Fig. 2 shows a simple example of a Simile model (designed to illustrate the use of Simile's visual language, rather than for its plausibility as a model). Combining our understanding of Simile's visual language with the labels used for the various components enables us to 'read' the diagram as follows:

> "The model consists of a fixed number of fields and a possibly-varying number of farmers (note the visual difference between the two submodels). There is an ownership association between farmers and fields (i.e. we know which farmer owns which fields). Each field can contain a grass and/or a crop. (We can not tell from the model diagram whether these are mutually exclusive, but we can see that the existence of each submodel—i.e. the presence of grass and the presence of crop—is conditional on the variable 'field type'.) The growth model for the grass is based on a single state variable, while that for the crop has separate state variables for the green biomass and the grain. Each field contains a multiple-layer soil water submodel, with a state variable (water content) for each layer. Grass and crop growth are dependent on aggregate soil

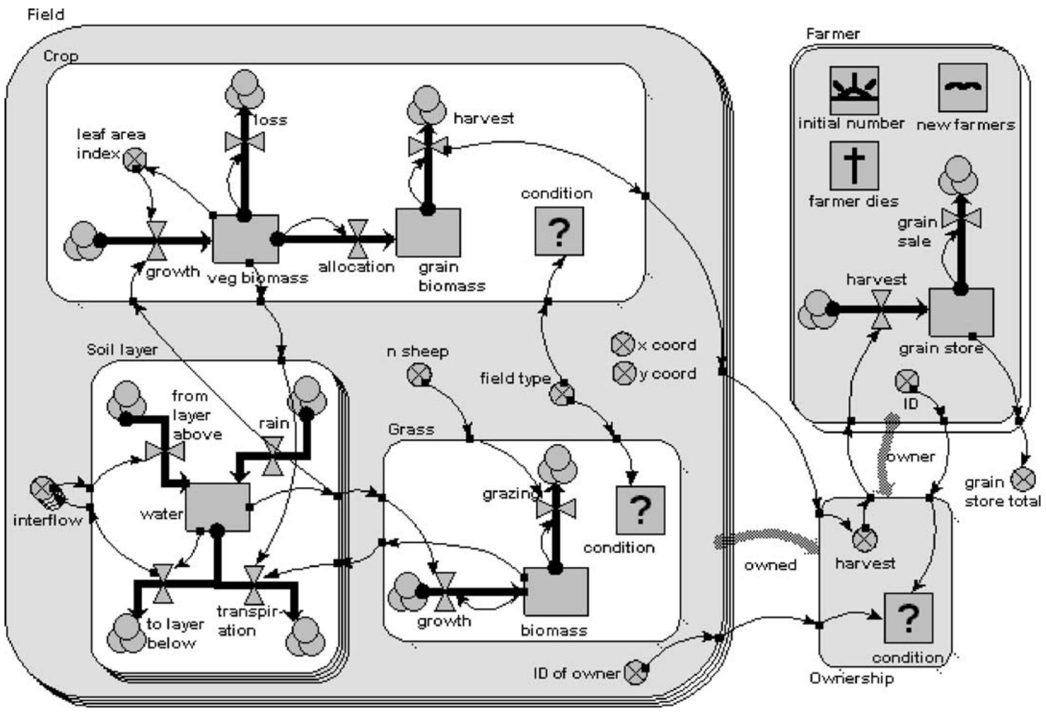

Fig. 2. An example Simile model diagram. See text for a verbal description of the model.

water, and in turn influence transpiration losses from the soil water compartment. Information on crop harvest is passed back to the farmers via the ownership submodel (so each farmer has access to the values on the grain harvest from the fields that he owns): this is used to build up the farmer's grain store, which is reduced by sales".

Note that we can produce this narrative simply by reading the diagram: we need no prior knowledge about the system. Note also that this diagram is not merely a presentational device. Rather, the diagrammatic notation, combined with the simple setting of properties for some submodels, contributes significantly to the mathematical structure of the model, just as (in a much more limited way) drawing a compartment-flow diagram tells a model interpreter what are the state variables, and what terms contribute to the rate-of-change expressions. This dramatically reduces the amount of information that the user needs to provide to complete the model: in this simple case, most elements needs only a value or a simple algebraic expression.

4. Simulating model behaviour

4.1. Running the model

A Simile model defines a set of differential equations. When a model is run, these equations are solved by numerical integration. Currently, we use simple Euler integration, but we intend to provide other methods.

In order to run a model (i.e. simulate its behaviour), the user simply selects a Run menu command, sets the simulation time settings in a Run Control dialogue window, calls up whatever displays are required, then clicks a Start button.

What's actually happening behind the scenes is that Simile generates a program in a procedural language, and it is this program which is executed. In each time step, the program first calculates the rate of change for all state variables, then updates all state variables. The program can be generated in Tcl or C++ (user's choice). Since Simile's visual interface is written in Tcl/tk, every Simile user automatically has Tcl installed, so this is the standard language to use. If Microsoft's Visual C++ is installed, or the user has installed the public domain Gnu C++ compiler, the any model can be alternatively run in compiled C++—with a speed increase of several hundred times.

Previously-constructed models can be run as stand-alone models, without having to load them into Simile and generate the Tcl or C++ program. The only requirement on the user is that they have previously installed Tcl/tk, since this is required for the visual interfacing.

4.2. Displaying the results of model behaviour

Simile provides a range of input/output tools (also know in Simile as 'helpers') for displaying the results of model behaviour. These include:

- a basic time-series display helper for showing the value of a model component over time. If that component has multiple values (for example, it is inside a multiple-instance submodel), then a line is drawn for each separate instance.
- a tabular display helper, which also enables tabulated values to be exported in.csv (comma-separated value) format for further analysis by e.g. a spreadsheet.
- map displays for grid-square and polygon spatial models.

The user is not limited to the display tools provided with Simile itself. Users can provide their own, either by editing the programs for the ones already provided or by writing new ones from scratch. This is possible because each display tool is actually a Tcl/tk program residing in a directory in the Simile installation. Since Tcl/tk is an interpreted language, each program is a text file, which can be edited. Moreover, each time you run Simile, it looks in this directory to see what files are there, and these provide the list of currently-available helpers. Therefore, all the user has to do is to add another file to this directory to get another helper added to the list.

5. Applications

Simile has been used to develop a wide range of demonstration models. These include models of:

- population dynamics;
- carbon, nutrient and water dynamics;
- animal movement;
- age- and size-class models of animal populations and tree stands;
- individual-tree-based forest models;
- models of evolutionary processes over multiple generations;
- spatial models of landuse change.

A sample of these models can be seen in the Model Gallery section of the Simile web site. The use of Simile for forest modelling is discussed in Muetzelfeldt and Taylor (2001).

In addition, Simile has been used in two major funded research programmes:

5.1. ModMED

The ModMED project (http://www.homepages.ed.ac.uk/modmed/) was concerned with understanding the dynamics of Mediterranean macchia (maquis) vegetation in response to fire and grazing, in order to inform the development of policy for the management of this type of vegetation. It involved the collection of data from historical sources, field studies and laboratory studies in Italy, Greece and Portugal. These data were used to develop Landlord, a hierarchical, GIS-based model of landscape dynamics, with individual pixels being modelled with community-level vegetation models. A variety of interchangeable community-level models were developed, including species-based and individual-based models.

Simile was used to develop the community-level models. An interface was developed enabling the user to specify the correspondence between the variables in the Simile model and those required by Landlord, and Simile's C++ program generator was modified to wrap up the generated program as a DLL with an interface compatible with that required by Landlord. Thus, any vegetation model developed in Simile could be exported to run within the landscape-level GIS.

5.2. FLORES

FLORES (http://www.ierm.ed.ac.uk/flores) is an international programme concerned with understanding the interaction between people and their natural resources at the forest margin in developing countries, in order to help in developing policy to improve their livelihoods. It has included the development of models in Indonesia, Zimbabwe and Cameroon through a collaborative process involving many different types of expert: local people and researchers in the social, ecological and agronomic sciences. The long-term aim is to develop models that can be used with some degree of confidence to explore the consequences of alternative policy options, but in the short term the major benefit of the FLORES programme has been the engagement of people who do not normally talk to each other in a process of participatory model development.

A typical FLORES model represents a very detailed view of rural communities. On the human side, the community is modelled in terms of a village submodel, containing multiple instances, one for each village. This contains a household submodel, represented as a population submodel to allow for changes in the number of households through immigration or breaking up on the death of adults. This in turn contains submodels for household-level demographics, household finances, annual (strategic) decision-making, and weekly labour allocation. The biophysical side is represented as a multiple-instance patch submodel, representing areas such as fields and forest clearings, and it contains submodels for forest dynamics, crop growth etc. A tenure submodel between households and patches is used to capture ownership and right-of-access relationships.

All FLORES models have been developed using Simile, for three main reasons. First, Simile's visual modelling interface has enabled people with little or no modelling experience to become involved in the modelling process, at least down to the level of commenting on the components and interactions in the model. Second, Simile has the expressiveness required for FLORES modelling, especially as regards the individual-based representation of households and patches, and associations between objects such as tenure. Third, Simile's ability to generate compiled C++ means that complex models, involving 100s of equations and 1000s of objects can be built which will run very efficiently.

6. Beyond Simile

Our aim in producing Simile is not to develop a monolithic modelling package. Rather, it has been to be part of a community developing tools for modelling in agro-ecological research. This ambition has one fundamental requirement: a common language for representing models, so that models can be shared between tools produced by many different groups around the globe. Since no such language currently exists, we have decided to use a public text format for saving Simile models to file, so that in principle any group can produce additional tools for processing Simile models.

In this section, we address two topics related to this theme. The first—the html generator—shows how it is possible to produce tools for processing Simile models completely independently of Simile itself. We then discuss the use of XML for representing Simile models.

6.1. Html generator for Simile models

The html generator is a program, written in the Prolog logic-based language, that can take any Simile saved model and generate a textual description of the model, marked up with html to render it suitable for viewing in a standard web browser such as Netscape or Internet Explorer. The model is presented submodel by submodel, with each submodel being described in terms of its compartments, flows, variables etc. The interactivity of html is exploited by hyperlinking all references to variables in equations: clicking on any of these jumps you to the part of the document where that element is defined. It is thus very easy to ‘browse’ around a model in a controlled, directed way.

The html generator program is written in such a way that it is very easy to change the amount of information that is produced. For example, it is very easy to modify the program so that it simply produces a hierarchical list of all the submodels—useful for a complex model with a dozen or more submodels.

6.2. XML representation of model structure

XML stands for the eXtensible Markup Language, a notation for marking up the content of documents. Html consists of a set of predefined tags, and marks up the appearance of documents for human readers. In XML, a community of people decide on what tags they will use to mark up documents, with a view to the documents being processed by computer programs rather than read directly by humans. For example, a group of ecologists might decide to use the tag ⟨species⟩ to delimit all species names in a document, and it is then possible for a program to retrieve all documents containing a reference to a certain species. XML is widely considered to be fundamental to the movement of content-rich documents across the internet, and to be a core technology for the Semantic Web, e-science and scientific Grids. XML documents can be viewed in current web browsers, there is a rapidly increasing number of tools for handling XML documents, and all the major programming languages have an application programming interface (API) for processing XML documents.

We have developed an XML Schema for representing Simile models. This has been designed for efficient processing by programs that wish to read or write model structure, while at the same time allowing the XML documents to be reasonably human-readable. A ⟨submodel⟩ tag is used to delimit each submodel, and these can be nested to reflect the nesting of submodels in the original model. There are tags for each of the Simile model elements: ⟨compartment⟩, ⟨flow⟩, etc, each containing further tags for the various attributes of each model element. The XML representation of any Simile model can be viewed in a web browser, and branches of the model can be opened and shut in the same way that one can use Windows Explorer to view a file directory system. Currently, the XML is generated from a saved-model file, but we intend to move towards the use of XML as Simile’s native model-representation format.

7. Discussion

The development of Simile has been motivated by three considerations: major problems with the current practice of modelling in agro-ecological research; the diversity of modelling paradigms that exist and usually considered as being mutually distinct; and the belief that computers can provide much more help for the modelling process than simply running simulations. In this section, we consider how well Simile addresses these issues, then how it compares with other approaches.

7.1. How simile addresses the problems with modelling practice

The problem of accessibility of modelling for non-programmers is addressed by having an intuitive visual interface. This encourages a two-stage approach to modelling: a diagrammatic conceptual stage, and a quantitative stage. In fact, one of the main impacts of Simile within the FLORES project has been its use for ‘red modelling’ (a term derived from the fact that the model diagram icons are red until a value or equation has been entered) in a participatory modelling process. The visual interface also greatly helps the problems of maintaining complex models, and in communicating models to others.

The difficulty of documenting models, and the danger that documentation no longer matches a model, is addressed by the fact that models are largely self-documenting—you do not need metadata saying how many state variables there are—and by attaching comments to model elements. The problem of (sub)model re-use is addressed by Simile’s ability to load submodels (which can be complete Simile models) from file, and by a flexible ‘plug-and-play’ mechanism. Similarly, the opposite problem—of testing submodels as independent models—is addressed by allowing any submodel to be saved to file as a full Simile model.

7.2. Simile's expressiveness

From an theoretical point of view, the most interesting aspect of Simile (indeed, any modelling language) is the choice of primitive model elements, especially for a language that claims to be able to cater for a range of modelling paradigms. Should there be fewer primitive elements?—it's hard to see how some modelling requirements could be expressed if we removed any (except for the 'initialiser' element, whose role could be performed by a migration element set at time zero). Should there be more? and, if so, is that because they are needed (i.e. there is a modelling requirement that is not satisfied by the current set of symbols), or because it would make life easier for the model designer? In fact, we are planning to introduce a 'memory' element: one that retains its value (and is thus a state variable) until explicitly set to a new value. This is certainly not needed—we can do the job with, for example, a compartment-flow structure—but would be more intuitive and parsimonious. What else should there be?

An interesting slant on this theme is to allow for a distinction between the primitive elements inside Simile, and those that the model designer has available. In fact, even now, Simile's internal language includes a 'function' element, and early versions actually showed a function symbol linked to every calculated variable. What you use now is in effect a shell around 'core' Simile—and it is possible to imagine other shells: e.g. for engineering block diagram notation, for chemical processes, etc. Fall and Fall (2001) argue for the development of domain-specific languages, giving an example of one they have developed for modelling landscape dynamics. We accept that reducing the conceptual gap between the modeller and the modelling language—allowing modellers to express familiar concepts directly—is desirable, but we argue that it is far better to do this as a shell around generic modelling software. This allows more experienced users to work at a lower level, and allows different domain-specific languages to be combined in a single model.

7.3. 'Computer modelling' is more than just running simulations

Currently, in agro-ecological modelling, almost all models exist on the computer as a computer program in a procedural programming language such as BASIC, Fortran or C/C++. As a consequence, most modellers consider that the role of the computer is restricted to running simulations.

However, there are in fact many aspects of the modelling process that computers can help with. The html generator, described above, is just one example, showing how the structure of a model can be displayed in a variety of ways. We could generate different program code from the same model, to simulate its behaviour on a lowly PC (at prototype stage) and then on a parallel computer. We could treat the model structure as a database, and search through for (e.g.) all variables influencing X, or all variables whose equation consists of a variable multiplied by a constant. We could search through a model catalogue for all models with certain characteristics—then do an automated comparison of model structure to identify similarities and differences between two models. We could automatically generate a textual description of a model, using canned phrases, and we could generate narratives explaining model behaviour by a program that has access both to simulation results and to model structure. However, what we do need to recognise is that all such tasks require that the model is represented declaratively (in the sense used in Section 2), as a set of symbols that have defined meaning in modelling terms.

7.4. Simile compared to other visual modelling environments

The main difference between Simile and other System Dynamics modelling environments used in agro-ecological modelling is Simile's ability to specify multiple instances of an entity. It may be considered to be a small difference: after all, most of the other environments have array variables, and some notion of submodel. However, the ability to draw a box around some part of the model and then specify many instances makes a huge difference: suddenly, spatial modelling and individual-based modelling become simple.

7.5. *Simile compared with object-oriented/component-based modelling*

Simile's comparison with object-oriented/component-based approaches (OO-COM for short) is rather more interesting. At one level, there's the fairly obvious difference that OO-COM involves programming, and Simile does not: for many people, that's a pretty significant difference. However, with the arrival of UML (Rumbaugh et al., 1999; Stevens and Pooley, 2000), and CAD tools (such as Rational Rose) (Quatrani, 2000) which not only support the production of UML diagrams but also generate partial or even complete code for UML models, this distinction will no doubt diminish in future.

There is a close correspondence between UML class diagrams and Simile diagrams with the influence arrows suppressed. In Simile, the structure of the model is represented by quantities and submodels grouped in container submodels and by associations between submodels. There is a direct mapping between this and a UML class diagram, with submodels corresponding to classes, Simile quantities corresponding to UML attributes, nesting of submodels in Simile corresponding to a composition association (contained class), and general associations having the same interpretation in both paradigms. Moreover, in UML new contained classes are often added to handle specific behaviours, just as submodels may be inserted into a containing submodel in Simile to handle certain tasks.

The influence diagram notation, fundamental to System Dynamics, is not present in UML. The similar data flow diagramming notation used to be used in software design (Rumbaugh et al., 1991), but is now deprecated. The collaboration diagram in UML is similar, but this shows the transfer of information between objects, not attributes, and thus is of much less use to the modeller.

However, the term OO embodies a particular conceptual framework, which differs in significant respects from that of Simile. First, in object-orientation the flow of information is conceptualised in terms of message-passing, whereas in Simile we think in terms of the influence relationship between variables. Second, it has a strong notion of data-hiding, restricting access to that subset of model variables which the programmer has chosen to make accessible. In Simile, in contrast, the user running a model has access to all model variables during the simulation, allowing any variable to be displayed and greatly helping in the analysis of model behaviour. Third, object-orientation generally involves a commitment to the concept of inheritance and specialisation hierarchies, whereas Simile has no such concept. It is possible this will be included in Simile in the future, but to date we have found that Simile's mechanisms for submodel re-use and adaptation are sufficient, and we doubt the feasibility of defining 'standard' taxonomies for ecology.

8. Conclusion

Simile demonstrates the feasibility of developing a visual modelling environment that has an intuitive user interface while at the same time having the expressiveness needed to handle a wide variety of model types and having the capability of generating computationally-efficient runnable versions of models. This approach offers a serious prospect of improving the efficiency and effectiveness of current practice in agro-ecological modelling—the more so if the further development and refinement of the approach becomes supported across the research community rather than being the activity of one group.

Acknowledgements

Simile was developed in projects partly funded by the UK's Department for International Development (DFID) under the Forestry Research Programme (Project Code R5652 and R7635), and in the ModMED project funded by the European Community's Environment Research Programme. The Simile software has been developed by Jasper Taylor (University of Edinburgh).

We acknowledge useful exchange of ideas with members of the 'modelling frameworks' group.

Appendix A: URLs

FLORES	http://www.ierm.ed.ac.uk/flores/
ModelMaker	http://www.modelkinetix.com/
ModMED	http://www.homepages.ed.ac.uk/modmed/
Powersim	http://www.powersim.com/
Simile	http://www.ierm.ed.ac.uk/simile
Stella	http://www.hps-inc.com/
Vensim	http://www.ventana.com/

References

Acock, B., Reynolds, J.F., 1997. Introduction: modularity in plant models. Ecol. Model. 94, 1–6.

Acock, B., Reddy, V.R., 1997. Designing an object-oriented structure for crop models. Ecol. Model. 94, 33–44.

Chen, J.-L., Reynolds, J.F., 1997. GePSi: a generic plant simulator based on object-oriented principles. Ecol. Model. 94, 53–66.

Costanza, R., Duplisea, D., and Kautsky, U. (Eds.), 1998. Modelling ecological and economic systems with Stella. Ecol. Model. 110: 1–103 (special issue).

Costanza, R., Gottlieb, S., 1998. Modeling ecological and economic systems with Stella: part II. Ecol. Model. 112, 81–247 (Special issue).

Costanza, R., Voinov, A., 2001. Modeling ecological and economic systems with Stella: part III. Ecol. Model. 143, 1–143.

de Wit, C.T., Goudriaan, J., 1974. Simulation of Ecological Processes. Center for Agricultural Publishing and Documentation, Wageningen, The Netherlands.

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

Forrester, J., 1961. Industrial Dynamics. Pegasus Communications, Waltham, MA, p. 464.

Grimm, V., 1999. Ten years of individual-based modelling in ecology: what have we learned and what could we learn in the future? Ecol. Model. 115, 129–148.

Johnson, I.R., Thornley, J.H.M., 1985. Dynamic model of the response of a vegetative crop to light, temperature and nitrogen. Plant Cell Environ. 6, 721–729.

Lemmon, H., Chuk, N., 1997. Object-oriented design of a cotton crop model. Ecol. Model. 94, 45–51.

Lorek, J., Sonnenschein, M., 1999. Modelling and simulation software to support individual-based ecological modelling. Ecol. Model. 115, 119–216.

McCown, R.L., Hammer, G.L., Hargreaves, J.N.G., Holzworth, D.P., Freebairn, D.M., 1996. APSIM: a novel software system for model development model testing and simulation in agricultural systems research. Agric. Syst. 50, 255–271.

Muetzelfeldt, R.I., Taylor, J., 1997a. The suitability of AME for agroforestry modelling. Agroforestry Forum 8 (2), 7–9.

Muetzelfeldt, R.I., Taylor, J. 1997b. The Agroforestry Modelling Environment. In: Agroforestry Modelling and Research Coordination, Annual Report 1996-97, ODA Forestry Research Programme, Project R5652. NERC/ITE Edinburgh.

Muetzelfeldt, R.I., Taylor, J. 2001. Developing forest models in the Simile visual modelling environment. Paper presented at the IUFRO 4.11 Conference on Forest Biometry, Modelling and Information Science. University of Greenwich, 25–29 June, p. 10. Available at http://www.ierm.ed.ac.uk/simile/documents/iufro3.pdf.

Porter, C.H., Braga, R., Jones, J.W. 1999. An approach for modular crop model development. Agricultural and Biological Engineering Department, Research Report No 99-0701, University of Florida, Gainesville, Florida, p. 15.

Potter, W.D., Liu, S., Deng, X., Rauscher, H.M., 2000. Using DCOM to support interoperabilty in forest ecosystem management decision support systems. Comput. Electronics Agric. 27, 335–354.

Quatrani, T., 2000. Visual Modeling with Rational Rose 2000 and UML. Addison-Wesley, p. 288.

Reynolds, J.F., Acock, B., 1997. Modularity and genericness in plant and ecosystem models. Ecol. Model. 94, 7–16.

Richardson, George P., Pugh, A.L., 1981. Introduction to System Dynamics Modeling with DYNAMO. MIT Press, Cambridge, Mass. London, p. 413.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., 1991. Object-Oriented Modelling and Design. Prentice-Hall.

Rumbaugh, J., Jacobsen, I., Booch, G., 1999. The Unified Modeling Language Reference Manual. Addison-Wesley, p. 576.

Sequeira, R.A., Olson, R.L., McKinion, J.M., 1997. Implementing generic, object-oriented models in biology. Ecol. Model. 94, 17–31.

Stevens, P., Pooley, R., 2000. Using UML: Software Engineering with Objects and Components. Addison-Wesley.

Timlin, D.J., Pachepsky, Y.A., 1997. A modular soil and root process simulator. Ecol. Model. 94, 67–80.

van Kraalingen, D.W.G. 1995. The FSE system for crop simulation, version 2.1. Quantitative Approaches in Systems Analysis Report no. 1. AB/DLO, PE, Wageningen.