Position Paper on

Declarative Modelling
in Ecological and Environmental Research

Robert Muetzelfeldt

Honorary Fellow
The University of Edinburgh
School of Informatics

Centre for Intelligent Systems and their Applications
School of Informatics
The University of Edinburgh
Appleton Tower
Edinburgh EH8 9LE
Scotland, U.K.

Email: r.muetzelfeldt@ed.ac.uk

Tel: +44 131 667 8938
Fax: +44 131 650 6513

Contents

Executive Summary

1. Simulation modelling is a key component of many ecological and environmental research programmes.

2. Most models are implemented in conventional programming languages, such as Fortran or C++.

3. There are, however, many problems with current practice. Models are time-consuming to implement and maintain, there is little reuseability of models, submodels or support tools (such as graphical output tools), there is no guaranteed correspondence between a model and its published documentation, and it is difficult for others to fully understand a model.

4. As a result, many groups have embarked on developing a modelling framework, to introduce sound software engineering principles into the modelling process. Most of these are based on a component-based methodology, such as Microsoft's COM (Component Object Model) technology. However, this approach still treats a model as something to be run (and therefore implemented as a program), rather than as a design to be represented.

5. I propose a vision of what the modelling process could be like, if we allow ourselves to think through the many activities involved in the modelling process.

6. Realising this vision depends on adopting a **declarative** approach to representing models. This means that models are stored as a set of data, defining the objects and relationships in a model, rather than as a program in a conventional (procedural or imperative) programming language.

7. To illustrate the concept of declarative modelling, an example is given of the same model represented in a variety of formats: English, mathematics, a computer program, a simulation language, a visual modelling environment, in a Microsoft Access database, in XML and in the logic-programming language Prolog. This shows that models can be represented declaratively, and hints at what becomes possible when this is done.

8. The technical issues involved in adopting a declarative modelling approach are considered. The key aspect is the separation of the representation of models from the tools developed for handling them (including running models - seen as just one way of handling them).

9. The representation of the model requires careful analysis of the concepts that the modeller should be able to use in designing a model. This constitutes the **ontology** of a language for representing models.

10. XML - the eXtensible Markup Language - provides the notation of choice for representing the **syntax** of a model-representation language. XML is widely used for publishing content (structured information) over the World Wide Web, enabling the information to be processed by software tools.

11. Special consideration is given to the development of software environments for running models and publishing them on the Web.

12. A proof-of-concept study is presented, based on the Simile visual modelling environment. The study is based on a spatial crop model, and demonstrates that it is possible to achieve the expressiveness and computational efficiency needed for research modelling using a declarative modelling approach.

13. The study also demonstrates that a declarative modelling approach enables the same model, represented in XML, to be transformed into a variety of formats (such as HTML), for display or analysis.

14. The potential use of a declarative modelling approach in a representative ecosystem modelling project is explored, in the form of a thought experiment. ATEAM is a current European project exploring the vulnerability of sectors relying on ecosystem services to global change, using existing forest, crop and hydrology models.

15. It is argued that the efficiency, effectiveness and integration within ATEAM (and similar projects) would be greatly increased if it took place in research culture designed around declarative modelling. The team would have virtually instantaneous access to a wide range of models, as well as a wide variety of tools for integrating the models, running simulations, and exploring the structure of the models.

16. In addition, a declarative modelling approach would greatly increase the participation of stakeholders in the modelling process.

17. The institutional aspects of adopting a declarative modelling approach are explored. There is the possibility that *de facto* standards will emerge through the piecemeal adoption of particular modelling environments, but that it is far better if the research community engages in a process of standards development. This requires an increased emphasis on funding the information technology infrastructure of ecosystem research.

18. The paper concludes with a critical evaluation of the declarative modelling approach.

1. Introduction

Computer-based simulation models are important tools in ecological and environmental research, and in the use of this research for policy formulation. The models enable often quite complex understanding to be combined in a single system, which can then be run forward through time to predict the state of the system in the future. By comparing such predictions with data, we can test our understanding of the underlying processes. By exploring how the responses vary depending on changing input values, we can explore the impact of external factors (such as climate change), and of management actions (such as a reduction in CO_2 emissions).

Currently, the great majority of such models are implemented as computer programs, written in a conventional programming language such as Fortran or C++. Overwhelmingly, the view in the research community is that the computer has just one role in the modelling process: it is to do the calculations needed to simulate the behaviour of the modelled system. Conventional computer programming languages are good at doing calculations: therefore, it seems reasonable to implement models as computer programs. This can be termed a **procedural** (or **imperative**) approach to modelling, since conventional programs specify a set of procedures to execute or (equivalently) specify a set of instructions to follow.

However, over the last couple of decades a number of groups have recognised that there are major problems with the current practice. These concerns have most frequently been expressed in terms of the lack of re-use or shareability of model components: different groups will implement their own soil water or crop growth submodel, rather than use an existing one, because of differences in programming languages or the difficulty of interfacing to somebody else's submodel.

However, the problems run much deeper than this. They include the time taken to build models, the difficulty for people to understand other people's models, the mismatch between model documentation and the implemented model, and the effort wasted in producing user interface and input/output utilities for individual models. These problems are not addressed by recent huge increases in computer power: they are to do with the way that models are built, not with the amount of power available for running models.

This paper presents a very different approach, called **declarative modelling**. It is based on the principle that models should be represented on the computer as a specification of the conceptual and mathematical structure of the model, not as a computer program instructing the computer to perform a set of calculations. The specification defines the objects and variables in the model, and the functional relationships between them.

Declarative modelling enables us to treat modelling as a **design process**, which is fundamentally what it is. Modellers decide what components and variables to include in their models, and how they should be connected. We can then develop computer-based tools to help with the process of building and working with models, just as CAD (computer-aided design) tools have emerged in other disciplines (architecture, electronics) to assist in designing buildings or electronic circuits.

A key feature of this approach is that we separate the <u>representation</u> of the design from the tools we develop for <u>processing</u> the design. In architecture, for example, the representation of the design is what is saved in a file when you have finished working with a CAD package: it contains statements defining the building (how big the walls are, what they are made of, etc). This same design can then be processed, as required, by a variety of tools: we could have one tool for showing the design in 3D, another for costing the building, another for working out heat losses. Similarly, a declaratively-represented ecological model is saved in a file using a language which is independent of any particular task (such as running a simulation) which we might wish to perform with the model. This file specifies, for example, the objects, variables and equations in the model. We could then have one tool for displaying the model structure in a certain way, another for simulating the behaviour of the model, and another for comparing the structure of two versions of the same original model. This allows us to develop tools for supporting many aspects of the modelling process, not just the running of simulations.

The distinction between declarative knowledge (knowing that) and procedural knowledge (knowing how) has long been recognised in human cognition. *I know that a bicycle has two wheels; I know how to ride a bicycle.* This difference is even ascribed to different parts of the brain, with the bilateral temporal lobes being used for semantic/conceptual knowledge, and the left frontal/basal-ganglia part being used for cognitive and motor skills (Ullman: http://slate.lang.uiuc.edu/lectures01.html).

Consider an IKEA wardrobe. We can describe it as a design - on paper, or in a computer-aided design package such as AutoCAD. The design says what bits we have, their properties, and how they are connected together. Or we can have instructions on how to construct it. The former is declarative, the latter procedural. The design is what was produced originally, to meet some specifications (size, cost, appearance, etc). And knowing the design, we can infer many things about the wardrobe: how big it is, how much it weighs, even possible procedures for constructing it. The instructions, on the other hand, serve a particular purpose - to get the flatpack into a functioning wardrobe - and are not an effective method for indicating what the wardrobe is actually like. If you simply gave the text instructions to someone, they would have a hard job making a drawing of what the finished wardrobe would look like.

Box 1.1 The procedural vs declarative distinction

This is a particularly opportune time for the research community to move towards a declarative modelling approach. First, there are a number of visual modelling environments, which prove the feasibility of having visual design tools, and simulation languages, which show that the mathematical structure of models can be represented in non-programming terms. Second, a number of groups are engaged in the process of designing a framework for their modelling activities in the field of environmental research, indicating a growing awareness within the community of the need to consider the way in which modelling is done. Third, there are similar initiatives in other domains, such as Modelica [see URLs[1] and Appendix 1] for physical systems modelling. Finally, the recent emergence of XML (the eXtensible Markup Language: see Appendix A2.1), a standard for the transfer of documents and data over the web, has spawned a large number of subject-specific markup languages. This makes it much easier to advocate the concept of a language for representing models, since there is now greater awareness of similar languages in other fields.

We distinguish between modelling and simulation. Declarative modelling is specifically concerned with the modelling process itself: how we design models, how we present them to other people, how we analyse their (mathematical) structure, how we can transform one model into another. It is not directly concerned with simulating the behaviour of models, including such activities as bench-marking, validation, sensitivity analysis, backcasting and parameter estimation. Such activities are, of course, vitally important, and declarative modelling can help, by ensuring that all models are fully compatible with tools developed for undertaking simulations. But this applies equally to models developed within, for example, the sort of component-based frameworks discussed in Section 3, so it is not central to the argument for a declarative modelling approach.

The aim of this paper is to consider the what, why and how of declarative modelling. What is meant by the term, and how does it differ from conventional approaches? Why should we consider adopting it? And how do we go about it - designing an appropriate language for representing models, building tools for handling these models, and putting in place the institutional structures required to make this work?

[1] [see URLs] refers to the list of URLs (web links) given after the References at the end of the paper.

2. The context: simulation models in environmental research and policy formulation

Large amounts of funding are going into environmental research, at the international, regional (e.g. European) and national levels. Umbrella programmes such as IGBP (International Geosphere-Biosphere Programme) and GCTE (Global Change in Terrestrial Ecosystems) constitute 'virtual organisations' responsible for initiating and co-ordinating research into a wide variety of issues concerning the environment, the impact of global change on mankind, and possible means of mitigating the effect of such change. The EU, through such initiatives as the Environment and Climate Programme in Framework 4 and the Energy, Environment and Sustainable Development Programme in Framework 5, has sponsored research projects into many key areas, including climate change, water, land-atmosphere interactions, desertification and biodiversity. The EU Framework Programme 6 continues these themes, with a greater emphasis on integration and the application of research results. At the national level, governments are funding both contributions to international programmes of research, as well as research at a more local level.

A key trend in this research is an increasing obligation for the research to be policy-relevant: that is, related to human concerns. For example, much of the research in the area of climate change, while still being undertaken as good science, is specifically concerned with informing government policy in response to e.g. the Kyoto protocol. As another example, results from projects in the area of desertification such as MEDALUS [see URLs] contribute to the United Nations Convention to Combat Desertification. The EU-funded ATEAM project [see URLs], which aims to determine the vulnerability of sectors relying on ecosystem services to global change, explicitly includes stakeholders in determining which key indicators of vulnerability should be considered, in order to ensure that the research includes policy-relevant indicators.

Computer modelling plays an important role in many of these research programmes. A variety of methodologies are employed, such as spatial modelling, Bayesian techniques, optimisation methods, and statistical modelling. However, by far the most important methodology is simulation modelling, which uses numerical methods to simulate process-based changes over time. These models are typically based on sets of ordinary differential or difference equations, representing the rates of change of key state variables, such as levels of carbon, nutrients, water, biomass or population size. Many of these models employ considerable degrees of disaggregation, dividing the state variables up into species classes, age classes, etc. Increasingly, the emphasis on policy relevance means that many of the models are linked to Geographical Information Systems (GIS), since many policy issues relate to the spatial distribution of resources, people, and climate change impacts. Finally, most of the models are made up from a set of submodels, reflecting the complex structure and interactions in ecosystems.

With few exceptions, models addressing policy-relevant issues are implemented in a conventional programming language such as Fortran, C or C++. For many people, making this point is superfluous: it is like saying "with few exceptions, cars have 4 wheels": that's what cars are, vehicles with 4 wheels. In the same way (they think), that's what computer models are: computer programs.

However, models are actually designs, and a simulation program is merely one way of inferring something from the design: the behaviour of the modelled system. There are languages for representing designs in other fields (electronics, architecture, engineering), so it is by no means obligatory to represent a model as a program: we can conceive of a design language for ecosystem models.

Furthermore, there are major problems with the implementation of models as computer programs (Box 2.1). These problems have been recognised by commentators over the last two decades or more: they severely limit the scientific value of the models, they make model development far more costly than it need be, and they prevent the development of tools to support the modelling process.

- Programs for research-grade models are costly to develop and maintain, often running into many thousands of lines of code and requiring specialist programming skills.
- Debugging is difficult. It is hard to put probes in a model to (e.g.) plot some arbitrary variable, and it is difficult to test submodels in isolation.
- Re-use of models, submodels and model support tools is difficult and thus rare.
- Programs, and thus the models they embody, frequently become obsolete at the end of the research project within which they were developed.
- It can very hard for others to comprehend a model from its program.
- There is no enforced correspondence between a model-as-program and the documentation (e.g. metadata, comment blocks or journal paper) that describes the model. Variables cannot have metadata attached to them.
- An equation in a conventional programming language is actually an assignment statement. This means that, while it is <u>possible</u> to write principled programs to implement a model, the language does not enforce this.
- There is a considerable conceptual gap between the constructs provided by a programming language, and those in the head of modellers when they design a model

Box 2.1 What's wrong with implementing a model as a computer program?

3. The development of modelling frameworks

3.1 Modelling environments and tools

For something like 3 decades, the need for some type of software support for the modelling process has been recognised, and various solutions have been proposed and prototypes developed (Rizzoli and Young, 1997).

In the 1970's, various large-scale models were developed for different biomes as part of the IBP - the International Biological Programme. See Patten (1975) for a summary paper on each biome. Several groups recognised major problems with the development of these models as large Fortran programs. Innis (1975) and Overton (1975) reported on the development of Fortran pre-processors (SIMCOMP and FLEX respectively) to aid in the development of models in their respective studies in the IBP.

Initial efforts in this area tended to fall into one of two main categories. First, several groups adopted a **simulation language**, most commonly one for continuous systems simulation. De Wit and his crop modelling group in Wageningen were the main proponents of this approach, using **CSMP** (the Continuous Systems Modeling Program) (de Wit and Goudriaan, 1974). CSMP subsequently died, though more recently **ACSL** (the Advanced Continuous Systems Modeling Language) [see URLs] has been used, notably by Thornley (1998) in developing the Hurley Pasture Model [see URLs].

The other main approach was to develop **Fortran-based frameworks** specifically tailored for modelling. Typically, this involved separating out the subroutines that held the model itself from those involved with support tasks (controlling the simulation, input of data and output of results). **FSE**, the Fortran Simulation Environment, (van Kraalingen, 1995) is a current example of this approach, which has been adopted by **ICASA** (the International Consortium for Agricultural Systems Applications) [see URLs] for in an initiative involving the re-implementation in a modular fashion of the DSSAT series of crop models.

Over the last decade or so, there has been a significant shift in the nature of the modelling frameworks proposed or actually used. Two main types can be recognised.

First, there has been widespread use of visual modelling environments, usually based on the **System Dynamics** paradigm (Forrester, 1971). System Dynamics is based on conceptualising a system in terms of compartments (stocks) and flows, and provides an intuitive way of modelling with differential or difference equations. **Stella** [see URLs]is the best known and probably most widely used visual modelling environment for System Dynamics: 3 special issues of the journal Ecological Modelling have been devoted to its use in ecology (Costanza *et al*, 1998; Costanza and Gottlieb, 1998; Costanza and Voinov, 2001). It is mainly used in an educational context, or for making relatively simple models, but van Noordwijk (1999) for example has used it for making a complex model (WANULCAS [see URLs]) of tree-crop-soil interactions involving several hundred equations. Although Stella has no explicit spatial-modelling capability, Maxwell and Costanza (1997) have linked Stella modelling to Geographic Information Systems (GIS) by using Stella to model a single patch, then using the exported equations to produce a program that is then coupled to the GIS.

Other similar packages include **Modelmaker**, **Powersim**, **Simile** and **Vensim** [see URLs]. Simile was developed under the author's guidance, specifically as a proof-of-concept demonstrator for declarative modelling. For this reason, extensive reference will be made to it in this paper, particularly in Sections 7 and 8.

The main characteristics of the use of visual modelling tools is that they tend to be used for whole-systems modelling: the model is essentially a flat structure, consisting of System Dynamics elements. Such modularity as there is typically for logically dividing a complex model up, rather than for building up a model from submodels.

3.2 Modular modelling, component-based approaches, and integrated modelling frameworks

In contrast, the second major current type of modelling framework is based on some notion of **modularity**. For a number of commentators, the need for modularity in model structure is the defining requirement for a modelling framework (Reynolds and Acock 1997: special issue of *Ecological Modelling*). It paves the way for the re-use of model components, enables a model to be constructed by a simple process of linking modules together, and allows for 'plug-and-play' use of alternative modules for a particular component of the system. (From the early days of modelling, the program implementing a model has often been split into subroutines/functions/procedures, corresponding to separate submodels. However, this approach typically provided little support for submodel re-use, or plug-and-play modularity.)

One way of handling modularity has been through the development of a modular modelling framework, in which modellers can select and link modules with no or little programming. A well-known example in the area of crop modelling is **APSIM** [see URLs] (McCown *et al*, 1996), which enables a cropping systems model to be constructed by selecting submodels for (e.g.) crop growth, soil nutrient dynamics and management to be plugged into a central simulation 'engine'.

Another approach for achieving modularity has been through the implementation of models in an object-oriented programming language, such as C++. Indeed, all the papers in the above-cited special issue of Ecological Modelling describe systems of this type. The approach has other claimed benefits for ecological modelling, including the analogy between composition and inheritance hierarchies in nature (a sheep has legs; a sheep is a type of mammal), and analogous constructs in object-oriented software design.

However, by far the most significant development in modular-based modelling in the last few years has been the adoption of a **component-based** approach as the way to organise the modelling effort within an organisation or in a large-scale project. A component is an independent software object that can be linked with other components. Its internal structure is hidden. It communicates with other components through the exchange of information across one or more interfaces.

Table 3.1 lists a number of such activities; there are doubtless numerous others. In addition, a more detailed description of two component-based frameworks - MODCOM and IMA - is given in Appendix 1.

A component-based approach can be applied at two levels:

- It can be applied at the level of decision-support systems (DSS), with individual parts of the system (the model, the user interface, the data management, the GIS, the results analysis) being separate components. Potter *et al* (2000) consider the design issues and choice of component-based technology for such systems in the context of forest DSS. In this view, the model is itself one component. This will not be considered further in this document: there is little to argue about, since a DSS is a large software product, and it is clear that component-based approaches offer a sound engineering approach to the development of such products.
- It can be applied at the level of the parts of a model, each part (submodel) being conceptualised as a component. For example, a cropping systems model might include 3 components: a crop growth model, a soil water model, and an insect pest model. It is this use of the term 'component-based approach' that interests us here.

A key characteristic of many such initiatives is the adoption of Microsoft's **COM** technology. COM (the Component Object Model) enables interoperability between modules developed in a wide variety of languages. Some use **DCOM**, which is a distributed version of COM (for operating over the internet). The principal alternative to COM is **CORBA** (the Common Object Request Broker Architecture), which works under both MS Windows and Unix operating systems, but is much less widely adopted. Raj (1998) and Chung *et al* (1997) provide an excellent technical comparison of the two technologies.

There are a couple of issues to consider about component-based approaches:

1. **1.** Why are there so many <u>different</u> initiatives to develop a component-based modelling framework? To an outsider, it seems slightly paradoxical that so many component-based modelling frameworks are being developed, when frequently a stated aim is to allow the re-use of submodels produced by various people. One would expect that a group intent on operating within a modular framework would, if possible, adopt that of another group, only developing their own if there was a need for some particular features that the others did not cater for. This would allow for maximum shareability of components, since the bigger the group, the bigger the pool of modules to share. As it is, there is little indication that components can be shared between different groups developing a component-based framework, and little in the way of initiatives to develop common standards to make such sharing possible. The only initiative that I am aware of that seeks to develop a common framework across several institutions is MODCOM [see URLs], and that is still limited in the number of participating institutions.
2. **2.** What common ground is there between a component-based approach and declarative modelling? This issue will be considered in depth in Section 11.2.

3.3 Critique of component-based approaches

As noted above, the great majority of modelling frameworks currently under development are founded on a component-based approach. There could well be some degree of convergence between component-based and declarative approaches in the future (see Section 11.2), but at the moment they are essentially alternatives: a group (or indeed the ecosystem modelling community) could opt for one approach or the other.

Component-based approaches are implemented by writing programs

At this point in time - and for the foreseeable future - models in implemented within a component-based framework are and will be implemented by people writing programs. This has two aspects.

First, and most obviously, it restricts the act of implementing a model to people with the appropriate skills. This ads to the cost of implementing models, and introduces a gap between the scientist who conceives the model and the software engineer who implements it.

Second, and more subtly, it reflects blinkered view of what modelling is about. As stated before in this paper, the main or only point in implementing a model as a program is to be able to simulate its behaviour. It is true that other people may be able to read the text of the program implementing the model, and thus check up on the detailed specification of the model. But if that's considered a requirement, then implementing the model as a program is definitely a poor way of helping other people to understand what it's like. By contrast, a declarative approach correctly views modelling as a design activity, and allows for the development of a wide variety of tools to support the design process.

The 'rate-then-state' issue

It is a principle of continuous-systems modelling that rates-of-change for all state variables should be calculated before any of the state variables are updated (van Kraalingen, 1995). Otherwise, artefactual behaviour can result from the arbitrary ordering of state variable updates. Some component-based modelling environments (e.g. MODCOM [see URLs]) correctly allow for this, but it can be very difficult to reverse engineer when legacy models (that is, previously-developed models) are adapted to fit into a component-based framework. And even when the modelling framework supports this technical feature, it can be hard to ensure that the programmers of individual components adhere to it.

Arbitrary influence linkages between components

When you construct a model in a visual System Dynamics modelling environment, you are at liberty to take influence arrows back and forth between submodels - there is no restriction on the linkages (apart from not introducing circularities). The model interpreter sorts the sequence of calculation of variables into the correct order. In some cases, however, such models could not be implemented in a component-based framework in which each submodel corresponds to one component: it is not possible to carve the model up into independent 'black box' submodels.

> *Example: consider a model containing a <u>farmer component</u> and a <u>crop component</u>, with crop harvesting being influenced by farmer labour input and influencing farmer revenue. The crop component cannot be called first, because it depends on labour input; but the farmer component cannot be called first, because it depends on crop harvesting.*

The only way around this is to split the 'rate' calculations in the component into several steps. However, the programmer needs to know the precise pattern of influence links in the external components in order to know how to do this. This violates a key principle of the component-based approach, that components can be programmed without regard to the other modules they may be combined with.

This may sound an esoteric criticism, but in fact we have found that a number of the submodels in the FLORES model [see URLs] have such influence linkages, leading to the view that this is a practical as well as a theoretical criticism.

Opacity and rigidity of the component interface

Paradoxically, the biggest weakness of the component-based approach is put forward by its proponents as its major strength. Components are delivered as black boxes: indeed, in COM they are provided in compiled binary form. The only way in which the outside world (other components or input/output tools) can communicate with them is through a fixed, pre-defined interface. This is claimed as a virtue because all that the user of the module needs to know is the interfacing: how the module does its job is irrelevant.

Now, this approach is fine in the software engineering world. If I want to use a text window component in a program, then it's great if all I need to know is how to call it. But models are designs - not just at the model level (in terms of components), but also at the inside-component level as well. Why deliberately hide the contents of a submodel from the modeller?

On top of this, imagine that the international ecosystem modelling community does actually standardise on a particular component-based approach. Scientists around the world start developing components according to the standard. Plant models, soil water and nutrient models, animal grazing models, and so on. Others want to start building ecosystem models from these components. What are the chances of these components being interfaceable: in other words, that the soil water output provided by the soil water submodel will match the soil water input required by the plant submodel? In a very controlled situation (a particular research team, having spent considerable effort defining the interfacing between the components of their ecosystem model), the components will indeed interface. But choosing components, developed without regard to each other, off the web, and one can virtually guarantee that they will not interface. In contrast, submodels in a declarative modelling approach do not have rigid, opaque 'shells', so it is easier to make connections between arbitrary pairs of submodels.

The conclusion is that a component-based approach might well work at a local level. But it cannot by itself be the way forward if we dare to envision a future in which we build models by choosing from a global pool of submodels, adapting them as needed to combine them together.

Table 3.1 A selection of component-based modelling initiatives

Title	Author	Reference	Comments
LMS: Land Management System	US Army Corps; Jim Westervelt (University of Illinois at Urbana-Champaign)	http://www.uiuc.edu	Emphasis on integrating existing spatially-explicit models. Considering HLA, DIAS or FRAMES for component technology
DEEM: Dynamic Environmental Effects Model	US Dept of Energy's Argonne National Lab, Chicago	http://www.dis.anl.gov/DEEM/hla.html	DEEM is an example application for providing integration and interoperability between environmental simulation models. Based on DIAS, an OO framework (based on DCOM) for integrating simulation models, information processing and databases. Models implemented in C++, SmallTalk.
	John Bolte, Oregon State University	http://biosys.bre.orst.edu/Pubs/Spokane/SPOKANE.htm	Advanced development of object-based modelling approaches, but not a single, deliverable system. More 'object-based' than 'component-based'
SME: Spatial Modelling Environment	Bob Costanza, Tom Maxwell, Ferdinando Villa	http://kabir.cbl.cees.edu/SMP/MVD/SME2.html Maxwell and Costanza (1997)	"Allows users to create and share modular, reusable model components." Applications: ELM (Everglades Landscape Model); PLM (Patuxent Landscape Model) Currently (1997), modules developed using in-house graphical modelling environment or MML (Modular Modelling Language), but SME wrapper being developed that will use Object Request Broker technology, for wrapping up legacy code. Also used to develop a number of TES ('Threatened and Endangered Species' models at GMS, the Geographic Modeling Systems lab (University of Illinois at Urnbana-Champaign): see http://blizzard.gis.uiuc.edu/HTMLdocs/TES.html
MMS: Modular Modelling System	George Leavesley	http://www.usbr.gov/rsmg/warsmp/mms	Includes tool for selecting and linking module input/outputs, and for linking to GIS (ArcInfo/GRASS). Modules implemented in Fortran or C. Applications: US Bureau of Reclamation river basin management for Gunnison River, Colorado; Tennessee Valley Authority/Electrical Power Research Institute river basin modelling; TERRA laboratory modelling of watershed-scale movement of nutrients and pesticides.
Modulus	RIKS	http://www.riks.nl/RiksGeo/proj_mod.htm	Combines modules land use change, hydrology, crops and natural vegetation. Based on ActiveX and COM.
NED-FVS:	Potter, Liu, Deng and		A full DSS, using DCOM.

	Rauscher		
LAMOS: Landscape Modelling Shell	Sandra Lavorel, Ian Davies, Ian Noble	http://www.gcte.org/lamos	A landscape modelling environment being developed within GCTE 2.2. Aims to permit comparison of alternative models, assemblage of new models from a toolbox of model components and user-produced modules. Emphasis on fire, vegetation spread and biogeochemical fluxes.
RiverWare	CADSWES: Center for Advanced Decision Support for Water and Environmental Systems	http://cadswes.colorado.edu/riverware/riverware_info.html	Visual, object-based environment for constructing models of catchments from components (pre-built or entered via an equation editor. Non-spatial, but does enable river networks consisting of multiple segments to be constructed.
HYDRA	CSIRO Land and Water; CSIRO Mathematical and Information Sciences; Sydney Water Corporation	Abel *et al*, (1994)	Integrates existing modelling systems in a 'federated' architecture for water quality management. Aimed at making it easy to add new and legacy models. Use can paint land use change scenarios.
EDSS: environmental Decision Support System	Modelling Center North Carolina	http://www.iceis.mcnc.org/EDSS	Aimed at air quality modelling, but designed to allow easy connecting of modules to make more complex models.
Forest Research Core Model	UK Forest Research		Framework for integrating a number of disparate legacy models
Modular Multi-purpose Integrated Assessment System	Tyndall Centre, U.K.	http://www.tyndall.ac.uk/research/theme1/theme1_flagship.shtml	Proposed framework for constructing climate change models from modules for individual subsystems (biosphere, cryosphere, hydrosphere, atmosphere, etc)
MODCOM	Oregon State University, Wageningen Agricultural University	http://www.biosys.orst.edu/modcom/	Framework based on COM for the assembly of simulation models from previously and independently developed component models
CMSS: Catchment Management Support System	CSIRO Land and Water	Davis and Farley (1997)	Effect of alternative catchment management practices on nutrient export. Customisable, transparent.

4. The vision: what modelling could be like

So, the current state-of-play is that numerous groups around the world are busy developing modelling frameworks, but:

- they are based on commitment to modules with rigid interfaces;
- they involve programming, mostly at both the level of making a model from modules and at the level of the internal structure of the modules;
- they see the implementation of the model on a computer as focussing on running a simulation;
- they do not see modelling as a design activity, to be supported by an appropriate CAD (computer-aided design) environment; and
- there is hardly any progress towards standardisation across groups.

It looks very much like the component-based approach is not the way forward. In fact, it seems to being sucking in lots of resources for purely local solutions. We need to take a different approach. The rest of this document is about just such a different approach.

I will begin by allowing ourselves to fantasise - to imagine that we can wave a magic wand, and think of what we'd really like to have.

Imagine if....

.... you could build a model without having to learn programming,

... the language you use for modelling would not restrict the design of the model

.... models could be built in a fraction of the time currently required;

.... you could run the same model on various platforms, even parallel computers;

.... models could be seamlessly integrated with GIS;

.... you could customise the types of displays used for visualising model behaviour;

.... you could automatically generate a description for any model in a variety of formats and level of detail;

.... you could automatically compare the structure of two models, to see their similarities and differences;

.... you could share models with other people as easily as you can now share Word documents;

.... you could search the internet for models just like you can now search for web pages;

.... you could edit or run any model you find in your web browser;

.... you could download any model you wanted by clicking on a link in a web page;

.... you could query the structure of a model, just like you can query a database;

.... you could insert any model as a submodel in some larger model;

.... you could extract a submodel from a larger model, and run it as a stand-alone model;

.... you could link model variables to data sets using metadata;

.... you could attach hyperlinks to any part of a model, taking a user to pages anywhere on the internet describing its biological or mathematical basis.

I hope to show that the fantasy is actually close to being realised. We can in fact treat the wishlist as an actual design specification, for a modelling approach that is well within the capabilities of current technologies. But first, I want to ruminate on what is meant by the term 'declarative'.

5. What does 'declarative' mean?

"We've retained our skills but we've lost our identities."
(Capt. Jean Luc Picard, Star Trek, from Conundrum, TNG-5)

An energy field sweeps through the Enterprise-D, producing retrograde amnesia in all the crew members. However, they all remember how to perform their duties! The episode dramatises the distinction between declarative and procedural memory systems in the brain. In addition, the episode provides a natural link to a discussion of patient HM, the famous but unfortunate man who surgery for epilepsy left him amnesic. HM - like the crew of the Enterprise - retained skills but lost his identity.

From: http://www.psy.vanderbilt.edu/faculty/blake/STIC/memory.html

The aim of this Section is to explore the meaning of the word 'declarative', preparing the ground for a more detailed examination of the term in the context of modelling in Section 6.

5.1 The 'declarative versus procedural' distinction

The term 'declarative' is usually used in contrast with 'procedural'. The distinction is well recognised in human cognition, with a distinction between 'knowing that' and 'knowing how'. I know that my cat is black; I know that the sun rises every morning. I know how to tie my shoe laces; I know how to bake a cake. The distinction is even taken down to different parts of the brain (Box 1.1).

When we move into the area of computing, the distinction corresponds closely to the distinction between data and program. Data describes what is; a program describes how to process data. A data set is a declarative representation of some set of information. A database containing information on the height and diameter of 1000 trees states something that is known about each tree. Data can be non-numeric: a catalogue of books, or the design of a building. A program, on the other hand, is procedural (also referred to as imperative): it describes the steps to follow (i.e. the instructions to execute) in order to achieve some goal. For example, we can write a program to work out the volume of each tree knowing its height and diameter. Table 5.1 compares the characteristics of declarative and procedural approaches.

We can think of this in terms of a boundary. On the left side of the boundary, we have the declaratively-represented information. On the right side we have a program, the instructions for processing this information. When we run the program, the information flows across the boundary into the program, is processed by it, and pops out the other side.

Declarative	**Procedural**
Describes what is	Says what to do
Data	Program
Requires separate problem solvers	Solves a specific problem
Can be manipulated and reasoned with	Can be executed
What	How
Sequence of statements (generally) not important	Sequence of statements is critical
Facts, concepts	Skills
Examples include databases, logic programming languages (e.g Prolog), functional programming languages (e.g. Haskell), designs, spreadsheets, models built in visual modelling environments, XSLT	Examples include programs written in conventional programming languages (Basic, Fortran, C, C++), control files for statistical packages
"diameter = 6.28*radius" defines a functional relationship between diameter and radius	"diameter = 6.28*radius" means "*calculate 3.14*radius, and assign the result to the variable called diameter*"
In humans, declarative knowledge is generally conscious, and expressed as words	In humans, procedural knowledge is generally automatic and subconscious, and expressed in actions
Declarative memory: facts, knowledge, events.	Procedural memory: how to do actions.

Table 5.1 Comparison of the characteristics of declarative and procedural approaches

5.2 Moving the declarative:procedural boundary

The distinction between declarative and procedural, between data and program, seems pretty obvious and barely worth discussing. What is <u>not</u> obvious - in fact, for many people is a novel and difficult concept - is that

the boundary between data and program is not fixed.

For any given problem, the software engineer can decide what should be represented as data and what should be programmed in a procedural language like Fortran or C++.

Consider a non-computing example. You're having a friend around for the evening. You could either give her instructions on how to get to your house from hers: <u>turn</u> left, <u>walk</u> 50 m,... (equivalent to a procedural program); or you could give her a map, and rely on her generic map-interpretation program (in her brain) to work out the route to take. You are replacing a specific set of instructions with some data (the map) and a more generic set of instructions (the brain's map processor).

Or take a computing example. In preparing the invitation, you could write a program which <u>prints</u> the text and <u>draws</u> some lines. **Or:** you could make a data file containing data like:

```
"text",10,20,"Please come around on Friday"  (There is some text at coordinates 10,20)
"line",10,25,50,25                           (There is a line from coords 10,25 to 50,25)
```

and write a generic program which is able to interpret the data in the file and produce the appropriate output for the printer. The specialised, one-off program that executes print and draw instructions is replaced by a more generic one, plus a data file containing some information.

We can venture the following principle:

> *Any given procedural computer program can be replaced by a declarative bit (data) and a more generic procedural program.*

Understanding this concept is absolutely fundamental to appreciating the concept of declarative modelling. This also involves shifting information (the model specification) from a procedural program (the way models are currently implemented) to data in a data file, to be processed by a more generic program.

5.3 Two examples of shifting the declarative:procedural boundary

We will now explore these concepts in a bit more detail, by considering 2 examples of moving the declarative/procedural boundary:
1 - a simple data-processing program;
2 - a simple function-evaluation program.
We will then consider <u>why</u> one should want to do this: what are the benefits of increasing the declarative content of a solution to a problem?

A simple data-processing program.

You'd probably be pretty horrified if you asked someone to work out the total of some numbers, and they came up with the following program:

```
a(1)=27.1; a(2)=53.2; a(3)=41.9...... a(157)=19.2
sum=0; for i = 1 to 157; sum = sum+a(i); next i
print sum
```

Why? Because this one program contains both the data as program assignment statements, and the instructions for analysing them. Instead, you would expect the data to be in one file (data1), and a **more generic** version of the program in another (prog1).

<u>File data1</u>

```
157
27.1, 53.2, 41.9, .... 19.2
```

<u>File prog1</u>

```
input n
for i = 1 to n; input a(i); next i
sum = 0; for i = 1 to n; sum = sum+ a(i); next i
print sum
```

Separating data and program gives huge benefits: the same data can now be analysed by different programs; and the same program can be applied to different data sets.

A simple function-evaluation program

Consider the following program, to work out the diameter of a circle from its radius:

```
input radius
diameter = 6.28*radius
print "radius="; radius; "  diameter="; diameter
```

Here, the data (well, datum: the value for radius) has already been separated out (as one would expect, in the light of the previous discussion).

But let's say that we actually had lots of functions that users might want - what should we do? Well, we can extend this program, adding in more and more functions, and providing some sort of menu to allow the user to choose the one they want. Or we could write many little programs, one for each function.

An alternative is to put the function itself, and the names of its input and output (independent and dependent) variables, into a text file (or a database), such as this:

```
"radius", "diameter", "diameter=6.28*radius"
"radius", "area",     "area=3.14*radius^2"
"radius", "volume",   "volume=4/3*3.14*radius^3"
```

Here, the 'data' are actually text strings (which would need to be enclosed in quotes in some systems, but not, say, in a database). Of course, there is no problem from the computing point of view in having non-numeric data: that's totally normal for databases holding information on people or books. The only thing that might cause a bit of surprise here is that one of the data items (the third one) is something that we are more used to seeing as a statement in a conventional program.

There is a big difference, however, between "diameter = 6.28*radius" in a program and in the database. In the former, it is an instruction (an assignment statement): "Evaluate the expression on the right-hand side of the equals sign, and assign the result to the variable on the left". In the database, it is simply a data value (as a text string): no more, no less. This is an important point: when we move on to consider declarative modelling, the procedural and declarative representations of a model might look similar, especially the equations, but they have fundamentally different meanings.

What can we do with this new approach? Well, an obvious requirement is to be able to calculate the value of any function in the database. There are several possible ways of achieving this. One, which would be easy to implement, is a generic program that asks the user for the required input and output variables, then selects the corresponding function, then generates a program behind the scenes, and then runs the program. For example, if the user enters:

```
Input variable:  radius
Output variable: area
```

then the output from the generic program could be

```
input radius
area = 3.14*radius^2
print "radius="; radius; "  area="; area
```

This is simply text output: the bits in bold (shown for explanatory purposes) only are picked up from the database; the plain text is canned text which is always the same. However, this text output is a program! - in fact, it could be the same as the program we could previously have written by hand. It can now be compiled and run, asking the user for a radius value and printing the area - then discarded, once it's done its job. We don't need to keep these programs around: all we need is the function database and the generic program-writing program.

Don't worry if you have trouble with the concept of a program-writing program. That is just one possible way of handling the database of functions. All that you need to accept is that, given a database of functions and some generic program that is capable of handling this, we can now do a job that previously involved writing a program.

Why do this? Well, we now have much more flexibility. We can easily add functions to our database without having to do work with a program. We could have different function databases (say, for physics, biology, mathematics...), and easily switch between them. We could work on elaborating the way we process functions (for example, plotting the function over a range of input values), confident that it would work for any function. But the real benefits become apparent when we consider the other things we can do with our function database, things that have nothing to do with actually evaluating them. For example, we could interrogate our database to find all functions that can calculate volume. Or we could print a catalogue of all the functions, one which is always up-to-date and 100% guaranteed consistent with the functions used to perform the calculations.

5.4 More examples of shifting the declarative:procedural boundary

We'll now consider in less detail a number of other examples of moving information from a procedural to a declarative form.

Electronic circuits

In the early days of computing, the behaviour of an electronic circuit was simulated by writing a program specific to the design of a particular circuit. Change the circuit, and you needed to change and re-compile the program. Then people developed CAD (computer-aided design) tools for electronics which enabled the circuit to be specified in terms of electronic components (resistors, capacitors etc, and how they are connected). The design of the circuit thus changed from being encoded procedurally in a program, to being represented declaratively as a circuit diagram. For example, the following could be 'data' for an electronics simulator:

```
resistor(r1,220).
capacitor(c3,100pF).
connected(r1,c1).
```

Again, it is obvious that it is far better to separate out the representation of the circuit from one particular form of reasoning we might wish to do with it. Once you have the data in this form, it is possible to simulate the behaviour of the circuit, as before. But it is possible to do many other things with the design: for example, prepare a layout for a printed circuit board; work out its heat budget; and calculate the cost of manufacture.

Spreadsheets

A spreadsheet is an example of a declarative data processing/modelling environment. Prior to spreadsheets, people wrote programs for working out their accounts and processing their data. In a spreadsheet, both the data and the calculations to be performed on it are held in one declarative representation: that's what you save to file. The only procedural part is now in the spreadsheet software itself, which is capable of processing the values in some cells using the functions in other cells to produce the results.

Logic programming

Appendix A2.3 presents a very brief introduction to Prolog, an example of a logic programming language. This differs from conventional (procedural, imperative) programming languages in that the 'program' is actually just a collection of facts and rules. It does nothing: it simply contains knowledge. For example, the facts could be about a road network, and the rules could be about how to reason with such information. Instead of 'running the program', one asks a question (e.g. "How do I get from A to B?"). The Prolog interpreter reasons with this collection of facts and rules to answer the query. Prolog is thus an example of a declarative programming language.

UML (Unified Modelling Language) and declarative programming

UML [see URLs] is a design methodology for object-oriented software engineering (Rumbaugh *et al*, 1999; Stevens and Pooley, 2000). It is based around a number of types of diagram, which jointly specify the software system being developed.

Originally, these diagrams were simply bits of paper that were given to programmers to guide the programming task. Companies developed UML modelling tools (such as Rational Rose: Quatrani, 2000) to help in drawing the diagrams. These software tools were extended so that they could generate skeleton code for the program: for example, to generate the text for all the classes, attributes and method wrappers. Some tools have the ability to reverse engineer: to produce the

UML design from code. In certain cases, the whole program can be generated from a complete UML description.

The UML story illustrates a marked trend in software evolution, from programmers writing procedural code to programmers writing declarative program specifications, which are then 'understood' by a suitable generic processor and turned into procedural code.

Topic	**Declarative bit**	**Processor**	**Non-declarative approach**	**Advantages of a declarative approach**
Data-handling using a program	The file containing the data	The program written to process the data	Write a program which contains the data in the program itself.	Separate data from program. Therefore same data can be analysed by different programs; same program can be used to analyse different datasets.
Mapping	The map	The human brain (able to find a path from A to B, for example)	Give instructions about how to get from A to B.	Map can be used to get from many points to many other points.
Electronic circuits	A computer-based representation of the circuit (resistor R1 connected to capacitor C3, etc)	Software tools able to: - simulate the behaviour of the circuit; - prepare a circuit layout for a printed circuit board; - calculate the cost of the circuit; - calculate the heat budget of the circuit.	Write a program for each circuit for each task. E.g. one program would be needed to simulate the behaviour of a specific circuit.	User constructs a circuit in intuitive terms (resistors, capacitors, etc, and their connections). The one design can be processed by multiple tools. Each tool can be used for all possible circuits, therefore huge savings in effort.
Spreadsheet	The spreadsheet itself (i.e. the spreadsheet it is when saved to file: the .xls file).	The spreadsheet software, e.g. Microsoft Excel.	Write a program to do the calculations.	Greater ease of use. Support tools (e.g. graphics; immediate update of values).
Logic programming	The logic 'program' (facts and rules) (see Appendix A2.3)	The logic programming interpreter, e.g. Prolog.	Difficult to conceive doing using conventional programming..	Only feasible way of tackling problems involving logic-based reasoning.

Table 5.2 Summary comparison of declarative and procedural software architectures

5.5 What are the benefits of greater declarativeness?

The declarative approach is generally closer to the way we think about the world. We usually separate knowledge about the world (as facts and rules) from things we can do with that knowledge (reasoning).

A declarative approach generally gives more flexibility. A procedural program (recipe) can usually be used for only one purpose. A declarative body of knowledge can be used in many different ways, to answer a wide range of different problems.

A declarative body of knowledge generally consists of independent fragments, each of which can be treated separately (e.g. in checking whether it is true or not). A procedural document needs to be treated as a whole, making it more error-prone and difficult to debug.

Declarative	**Procedural**
Use knowledge in multiple ways	Use knowledge one way
Low efficiency (though not necessarily true)	High efficiency
High modifiability	Low modifiability
High cognitive adequacy (close to the way we think)	Low cognitive adequacy (less intuitive way of thinking)
Higher level of abstraction	Lower level of abstraction
Independent collection of facts	A composite whole, difficult to de-compose.
Easy to validate	Hard to debug
Transparent	Black-box

Table 5.3 Relative merits of declarative and procedural approaches

5.6 Conclusion

In many areas, a declarative approach represents a real and increasingly-used alternative to approaches based on conventional programming. It offers significant benefits, in terms of increased flexibility, transparency and intuitiveness.

In the following section, we will begin to explore its relevance to modelling, by comparing the procedural and declarative representation of a simple ecological model.

Sources
Some points relating to the declarative:procedural divide have been taken from the following sources:
http://www.uiowa.edu/~c07p200/handouts/declarative_procedural.html
http://computing.unn.ac.uk/staff/cgpb4/prologbook/node11.html
http://ai.eecs.umich.edu/cogarch2/prop/declarative-procedural.html
http://www.cs.rochester.edu/u/kyros/Courses/CS242/Lectures/Lecture-14/tsld003.htm
http://www.cs.cornell.edu/Info/Projects/NuPrl/cs611/fall94notes/cn2/subsection3_1_2.html
http://www.psy.vanderbilt.edu/faculty/blake/STIC/memory.html
http://www.adass.org/adass/proceedings/adass00/reprints/O6-03.pdf

6. Declarative and procedural representations of a model

The following walk-through presents a variety of ways in which one model can be represented. The aim of doing this is to introduce some basic concepts, and to prepare the ground for a more in-depth consideration of declarative modelling in subsequent sections.

All the methods relate to the same model. This is a very simple ('toy') model, based on a couple of differential equations. The same principles apply to 'real' models, but the example here has been kept deliberately simple (indeed, simplistic) in order to focus attention on the different forms of representation.

6.1 Verbal description

We begin with a verbal description of the model (**Box 6.1**). This is rather typical of the sorts of statements that one might use in outlining a model to a colleague, or in the "Model description" section of a journal paper about the model.

Box 6.1 Verbal description of the model

"The model has two components, a grass component and a soil water component, each represented by a single state variable. Grass growth is assumed to be positively related to grass biomass and soil water content. Soil water content increases through rainfall, and decreases by transpiration, which depends on both grass biomass and soil water content."

There are a number of interesting things to note about this example.

First, as with many published model descriptions, it does not give a complete mathematical description of the model. The test that we can apply is: could an experienced programmer/ modeller re-implement the model from the verbal description? Here, for example, we do not know the precise nature of the functions for grass growth, transpiration or rainfall, so the answer has to be "No". Frequently, of course, published descriptions of this type will include equations for key relationships, but nevertheless it is fairly rare for such verbal descriptions to permit complete re-implementation of the model.

Second, despite the previous point, the description does permit an experienced programmer/ modeller to at least make a start on writing a program that can simulate the behaviour of this model. The program needs an input section for the initial values of the two state variables (grass biomass and soil water content); a loop over time; dummy equations for working out the rate of each process (until we have more information); and assignment statements for working out the net rate of change of each state variable and for performing numerical integration. If the verbal description had included more information - equations for the 3 processes, or perhaps a verbal statement saying that "grass growth is proportional to both soil water content and grass biomass" - then the program could be completed and would require only numerical inputs to enable simulation to take place.

Third, and most important: the verbal description is a **declarative** representation of the model. It simply tells us what the model is like. It does not in itself tell us how to calculate changes in grass biomass. Knowing how to perform these calculations - either manually or by writing a computer program - requires the ability to **interpret** these statements correctly. In the case of a verbal description such as above, this interpretation would have to be by a human - our standard "programmer/modeller", who knows both modelling concepts and how to program. At this stage of computer-based natural-language understanding, we could not envisage a computer doing this. But, as we shall see below, it is perfectly feasible for a computer to interpret a declaratively-represented model, provided we use a rather more formal language for representing the model.

6.2 Mathematical description

The model can also be represented mathematically, as a pair of ordinary differential equations (**Box 6.2**).

Box 6.2 The model represented using mathematical notation

$dG/dt = k_1 G W$

$dW/dt = R - k_2 G W$

where:

G is grass biomass

W is soil water content

R is rainfall

k1, *k2* are parameters

This is a more compact, formal and precise statement of the model. It indicates clearly that we conceptualise the system as one of continuous change, and it shows the precise mathematical nature of the grass growth and transpiration terms.

However, the essential difference between the two forms of representation does not really relate to their ability to convey the complete specification of a model. The mathematical description could be incomplete: we could, for example, state that

$dW/dt = R - f(G,W)$

Conversely, we could arguably produce a complete and unambiguous verbal description of the model. Rather, it relates to the increased formality of the representational notation. It is this which makes the notation more compact. It also reduces ambiguity. And it also lends itself to the automated processing of the model structure, as we shall see in Box 6.4 below.

6.3 Representation of the model as a computer program

We can write a computer program whose job is to solve the model - i.e., to work out the values of the state variables at some future point(s) in time.

For simple models, the solution of the model can be performed **analytically**. This means (in the present example) that we can derive equations giving the value of grass biomass and soil water content at any future point in time. In this case, the computer program would consist of just two equations, with inputs of the initial values for grass biomass and soil water content, the rainfall value, the value for the two parameters, and the value of time we wished to solve the model for. However, this is only possible for ecologically-trivial models.

In the vast majority of situations, we need to solve the model through **simulation**, by numerical integration of the underlying differential equations. In essence, this involves cycling through a loop in which we work out the changes to the state variables over some small period of time, adding or subtracting these changes to/from the state variables, then repeating the process with the new values for the state variables. **Box 6.3** presents a program for simulating the behaviour of our example model. This happens to be in BASIC, but the same program could be implemented in any conventional procedural programming language (such as Fortran or C).

Box 6.3 Representation of the model as a program in a procedural programming language

```
input grass, water                                 Input initial values for state variables,
input rain                                         rainfall, and parameters
input k1, k2

for day = 1 to 365                                 Loop over days
  for timestep = 1 to 100                          Loop over small time step
    growth = k1*grass*water                        Work out processes
    transpiration = k2*grass*water
    grass_rate_of_change = growth                  Work out rates of change for state
    water_rate_of_change = rain - transpiration    variables
    grass = grass + 0.01*grass_rate_of_change      Update state variables
    water = water + 0.01*water_rate_of_change
  next timestep                                    End time-step loop
  print day, water, grass                          Print current state every day

next day                                           End loop over days
```

The key thing to note about this example is that this is not in fact a **description** of the model. Rather, it is a set of **instructions** that a computer should follow in order to enable a certain type of inference to be drawn from the model, using a particular numerical approach. The inference relates to the future values of water and grass given their initial values and the value of parameters. The particular numerical approach is that of Euler integration.

This is by far the most common method for expressing ecological models. There are many problems with the use of this approach as the primary method for implementing ecological models (see Section 2). There are also many variations on this approach, such as the use of program modules, component-based methods and object-oriented methods (see Section 3), complicating the task of developing a simple analysis of the relative merits of a programming-based approach.

6.4 Representation of the model in a simulation language

A simulation language enables the mathematical relationships of a model to be expressed without having to program the procedures for numerical integration, input/output, etc.

There are simulation languages for a variety of modelling approaches. We will restrict our attention here to those designed for handling systems of ordinary differential/difference equations, since they are by far the most widely used in the ecological field.

Box 6.4 shows the model implemented using ACSL, an industrial-strength simulation language used in some ecosystem modelling projects (e.g. Thornley, 1998).

Box 6.4 Implementation of the model in ACSL

```
PROGRAM Simple grass-water model
CINTERVAL cint = 0.1
NSTEPS nstep = 1
CONSTANT rain = 100, k1 = 0.1, k2 = 0.1
CONSTANT grass_init = 100, water_init = 100
growth = k1*grass*water
transpiration = k2*grass*water
grass_rate_of_change = growth
water_rate_of_change = rain-transpiration
grass = INTEG(grass_rate_of_change, grass_init)
water = INTEG(water_rate_of_change, water_init)
TERMT(t .GE. 20)
END
```

We note strong similarities with both the representation of the model in terms of differential equations (**Box 6.2**), and indeed with parts of the implementation of the model as a computer program (**Box 6.3**). The simulation language enables one to concentrate on expressing the mathematical relationships in the model, and leave the procedural bits - such as how to undertake numerical integration - to the software package.

The programming approach (Section 6.3) is an example of a procedural approach to modelling. The use of a simulation language, on the other hand, exemplifies a declarative approach, in which we simply express the mathematical relationships that apply, and leave it up to some modelling-aware interpreter to process these in the correct way. You may well be thinking that there is not a lot of difference between these: after all, the model equations are expressed in identical syntax, and both require learning some form of text-based computer language. However, there are several key differences:

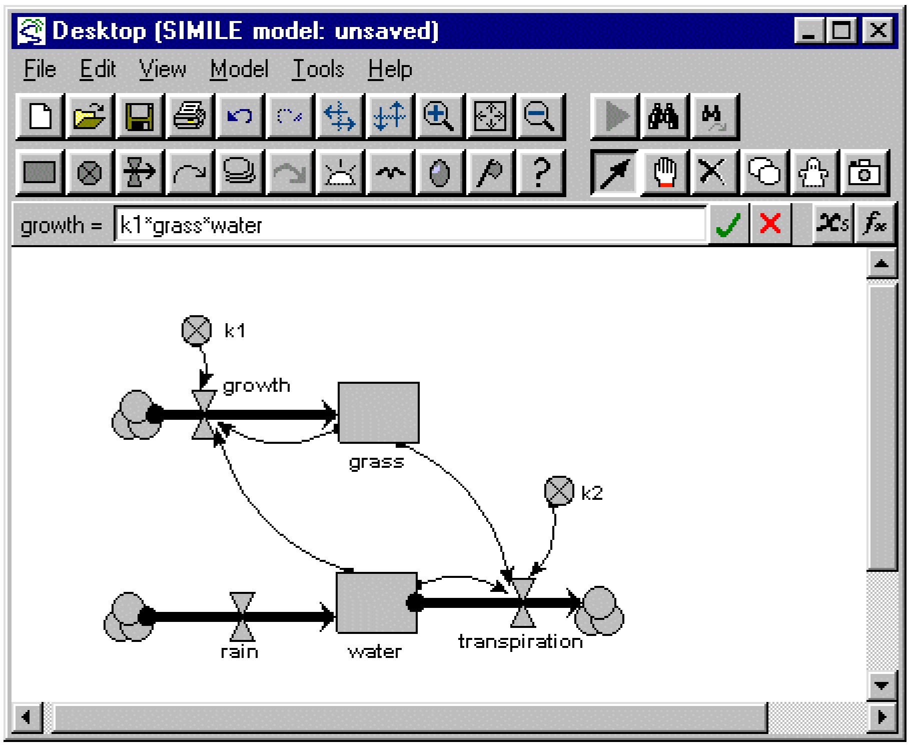
In the procedural program, the statements must be read in order, starting from the first. In the simulation language, the statements can be entered in any order: they are sorted into the correct order for evaluation by the simulation language interpreter.

In the procedural computer program, each equation is actually an instruction, called an 'assignment statement'. The correct way of reading it is: *Work out the expression on the right of the equals sign, then assign this value to the variable on the left.* In the simulation language, on the other hand, the equation is a statement of the mathematical equality between the left-hand and right-hand sides. One consequence of this is that it is not an error in the programming language to have two separate statements with the same variable on the left-hand side, though this would be an error in the simulation language.

6.5 Visual modelling environments

Graphical user interfaces (GUIs) have revolutionised the ease with which ordinary people can access complex computing resources. In the field of simulation modelling, a number of software packages have been developed to enable users to build models without having to learn a conventional programming language or the syntax of a simulation language.

A visual modelling environment enables a model to be constructed diagrammatically, as node-and-arc diagrams. Icons representing particular model elements (nodes) are selected from a toolbar and placed in a drawing window. Interconnections between model elements (arcs) are made by drawing lines from one to the next. The following example (**Fig. 6.1**) shows the grass/water model implemented in the Simile visual modelling environment (described in detail in Section 8).



growth = k1*grass*water

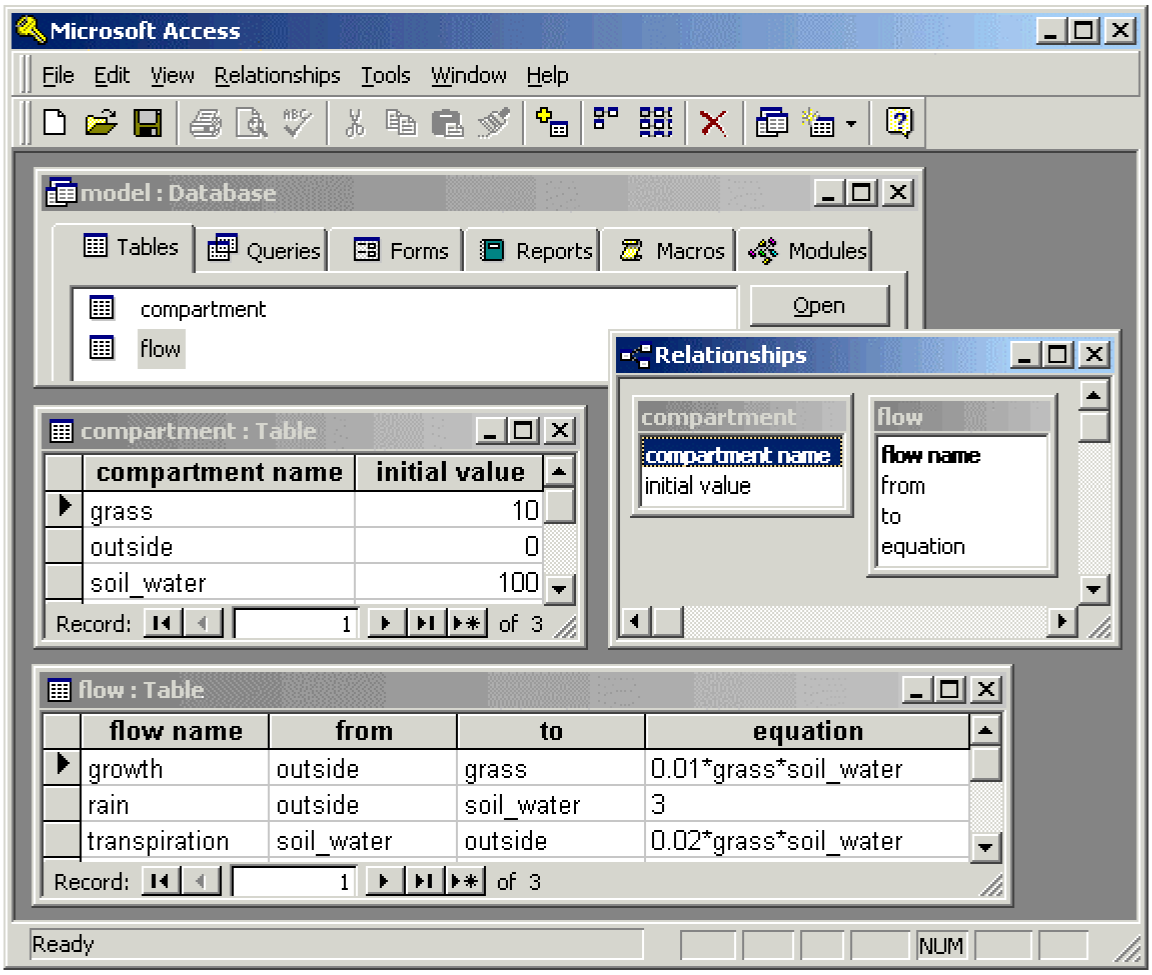
Fig 6.1 Representation of the grass/water model in the Simile visual modelling environment.

6.6 Representation of the model in a database

Represent a model in a database?! Seems like a strange idea, but consider the following example implemented in Microsoft Access (**Fig. 6.2**).

Here we have a relational database with two tables: 'compartment' and 'flow'. The 'compartment' table has two fields: 'compartment name' and 'initial value'. The 'flow' table has 4 fields: 'flow name', source and destination compartments for the flow ('from' and 'to'), and the flow equation. This represents a complete specification of the model: an appropriate human (or computer-based) interpreter could write a program to simulate the behaviour of the model so defined. Though a simple example, it could be easily extended to handle parameters, intermediate variables, submodels, etc.

But - it's in a database! So we could also do normal database-type operations: interrogate the structure of the model ("tell me the equations for all flows associated with the compartment 'water'"); or print out a summary report on the model (number of compartments, number of outflows and inflows). If we add a 'model ID' field to each table, then we could store information on many models in a single database, put it onto a web-enabled database, and enable anyone to retrieve models according to their structure, through their web browser. In other words, we can now do many things with the model that we could not do if it were implemented merely as a computer program in a conventional programming language.



compartment name	initial value
grass	10
outside	0
soil_water	100

flow name	from	to	equation
growth	outside	grass	0.01*grass*soil_water
rain	outside	soil_water	3
transpiration	soil_water	outside	0.02*grass*soil_water

Fig. 6.2 Representation of the model in an Access database.

6.7 Representation of the model in XML

XML stands for the eXtensible Markup Language. It is actually a metalanguage - a form of notation within which a wide variety of languages can be defined. XML is like HTML - the hypertext markup language - in that items of text are enclosed between start and end tags. However, unlike HTML, the tags in XML relate to content rather than appearance. Also, the tags in XML are not pre-defined - it is up to any community of people to define their own set of tags. XML is already widely used, in commerce and industry, by news agencies, and in many e-science initiatives. There are now hundreds of markup languages developed in XML. See Appendix A2 for more information on XML and the related technology of XSLT - the eXtensible Stylesheet Language for Transformations.

Given that a major rationale for declarative modelling is to enable models to be transferred over the internet; and given that XML has been specifically developed for transferring information over the internet, it is natural to explore the use of XML for representing models. We will discuss this in depth in Section 8 (proof-of-concept using Simile). Here, I will illustrate the idea by presenting our simple model marked up in XML.

Box 6.5 Representation of the model in XML

```
<?xml version="1.0"?>
<model>
    <compartment>
        <name>grass</name>
        <initial_value>100</initial_value>
    </compartment>
    <compartment>
        <name>water</name>
        <initial_value>100</initial_value>
    </compartment
    <flow>
        <name>growth</name>
        <to>grass</to>
        <equation>0.01*grass*water</equation>
    </flow>
    <flow>
        <name>rain</name>
        <to>water</to>
        <equation>3</equation>
    </flow>
    <flow>
        <name>transpiration</name>
        <from>water</from>
        <equation>0.02*grass*water</equation>
    </flow>
</model>
```

This XML **document** contains a single **root element**, model, enclosed between the <model>...</model> **tags**. This particular model contains two compartment elements (enclosed between the <compartment>...</compartment> tags), and 3 flow elements (enclosed between the <flow>...</flow> tags. Compartments are characterised by a name and an initial value; flows by a name, the name of the compartment they come from and go to (each optional), and an equation.

XML is a powerful and well-supported technology, making it the method of choice for the representation of structured information. There are many tools available for validating XML documents (making sure that they conform to the syntax of the specified language), for displaying XML in a wide variety of useful ways, and for transforming XML documents into other forms of representation (e.g. software-specific notation for representing models). Section 8 gives a taste of what is possible.

6.8 Representation of the model in Prolog

Prolog ("Programming in logic") is a computer language that was developed for representing and reasoning with statements in a particular form of logic, called 'first-order predicate logic'. It represents an approach to computer programming that is totally different from that of most computer languages. In Prolog, a computer program consists of a collection of statements, which are taken to be true statements about some subset of the world. These statements can either be facts (*rabbits eat grass*) or rules (*X is a herbivore if X eats grass*). An introduction to Prolog is given in Appendix A2.

Why is this relevant to declarative modelling? First, declarative modelling involves making lots of factual statements about a model: "The model contains a compartment called biomass and a flow called photosynthesis", for example. Prolog provides a compact way of capturing these statements. Second, a lot of the work we do as modellers involves reasoning about models: following influence chains through the model, interrogating the model structure, or finding models with certain properties. Some of these operations can be done in XSLT, but Prolog rules provide a far more powerful mechanism for doing this type of reasoning with models. Prolog is thus an ideal language for representing models declaratively (as a set of facts), and also for reasoning with models so represented. Muetzelfeldt *et al* (1989) discuss the utility of Prolog for ecological modelling; Robertson *et al* (1991) report on research into the representation of ecological models as logical statements in Prolog; and McIntosh *et al* (2003) presents a rule-based model, implemented in Prolog, for vegetation dynamics.

Box 6.6 shows one way of representing the grass/water model in Prolog, using System Dynamics concepts.

Box 6.6 The grass/water model represented in Prolog

```
compartment(grass, 10).
compartment(water, 20).
flow(growth, outside, grass, 0.01*grass*water).
flow(rain, outside, water, 10).
flow(transpiration, water, outside, 0.02*grass*water).
```

We state here that the model has 2 compartments and 3 flows. The 'compartment' predicate has two arguments, representing the compartment name and its initial value. The 'flow' predicate has 4 arguments, representing the flow name, the source and destination compartments for the flow, and the equation used for calculating its value. Note the close similarity to the relational database representation: in fact, Prolog can be considered as a relational language.

Prolog can be used for interrogating and reporting on model structure, in much the same way as using XML and XSLT, or the model in an Access database. However, the real power of Prolog comes when we want to do some form of reasoning with the model design, going beyond interrogation or reporting. For example, we might want to:

- trace through influence or flow networks, for example finding all flows directly or indirectly influenced by a certain parameter, or all compartments on a particular flow network;
- check model syntax, i.e. that the model design adheres to rules of System Dynamics (e.g. no flows pointing to variables);
- generate a C++ program from the model representation - this is exactly what Simile does when you want to run a model;
- transform a model - e.g. collapse a complex part of the model into something simpler (Muetzelfeldt and Yanai, 1996);
- automatically compare two models - perhaps each derived from a parent model by two separate groups over a 6-month period.

Such tasks can also be programmed in other languages, but Prolog is especially suited for it: it is a symbolic reasoning language, and thus ideal for reasoning with a symbolically-represented model.

6.9 Conclusions

In this section I have attempted to show that the same model can be represented - on a computer - in a variety of ways, procedural or declarative. The declarative ways are largely interchangeable - we can convert from one to the other to exploit whatever benefits come from a particular form of representation. In general, we can convert into but not out of a procedural representation (as a computer program). This is because the formal declarative notations are all expressed in terms of mathematical or modelling concepts, whereas the procedural notation is expressed in terms of programming notation, and we cannot - in the general case - guarantee inferring the model design from a computer program.

Declarative modelling is not new. As mentioned above, it forms the basis for simulation languages (such as ACSL) and visual modelling environments (such as Stella, Modelmaker, Powersim, Vensim and Simile), all of which enable a model to be constructed without procedural programming, and all of which enable a model to be saved to file as a design specification, not as a program. Modelica (Appendix 1) is a declarative modelling language developed for physical systems modelling. SDML is a Strictly Declarative Modelling Language developed for agent-based modelling (Edmonds *et al*, 1996), while Esteva *et al* (2002) describe a logic-based language for modelling multi-agent systems. Previous work on the use of Prolog for declarative ecological modelling is mentioned in Section 6.8 above.

However, this paper differs in emphasis from these other approaches. It places the declarative representation at centre stage, as something which can allow the same model to be represented in a variety of ways (as exemplified in this Section), and to be processed in a variety of ways, as we will see in Sections 8 and 9. And it proposes that a declarative approach can bring about a radical change in the practice of modelling within a large domain: that of ecological and environmental research.

In the following section, I will analyse in rather more detail the issues involved in developing an architecture for a declarative modelling approach.

7. Architecture for a declarative modelling approach

The key to realising the vision for environmental modelling is this: to think of a model as a design to be represented and processed rather than as a program to be run. This is the essence of the declarative modelling approach.

Realising the vision thus depends on having two types of object: **models**; and **modelling tools**. The model will be represented in some language; the tools need to be capable of reading, processing and/or writing models represented in that language.

It is important to reiterate that, by its very nature, a declarative modelling approach has no software core, no software package that users must install if they are to work with the approach. If a community of ecosystem modellers commit to such an approach, the <u>*only*</u> *thing that they are committing to is a specification of the model-representation language. In effect, they are committing to a standard, expressed in a standards document. They can then obtain and use tools that adhere to this standard, but no individual tool is fundamentally required. There may be a variety of tools available for building models; a variety for generating runnable versions of the model and running simulations. In the near future, these tools will be available over the web, as web services, so they won't even need to be installed on one's own computer. It is a highly distributed and robust approach.*

Section 7.1 considers the specification of requirements for a declarative model-representation language. Sections 7.2 to 7.4 consider the development of an appropriate language: what elements should the language be built upon? In Section 7.5 we consider the development of tools for handling models represented in this language, while in Section 7.6 we consider one particular type of tool - tools for simulating the behaviour of the declaratively-represented model. Finally, Sections 7.7 and 7.8 are concerned with assembling a variety of tools together in a modelling environment, and with publishing models on the web.

7.1 Requirements specification for a model-representation language

We need to define the scope of the language - what range of models should the language be able to represent? At one extreme, we could restrict the language to sets of differential-algebraic equations (DAE's). At the other, we could aim to include every type of model that has ever been used in ecosystems research. Clearly, the more restricted the scope is, the easier it is to devise a suitable language - but the less use it will be.

The **Table 7.1** lists part of the requirements specification for the Simile visual modelling environment (see Section 8). Some parts of the specification are expressed in the sort of terms that modellers use to describe their models (e.g. "age-class model"). There is some overlap between these terms, and equally well there are some other terms that modellers would use that are not mentioned here. This does not matter too much, provided that the eventual design is capable of handling the range of requirements. Other parts (such as modularity) refer to more abstract features of the modelling language.

Note that, as we shall see in the next subsection, there is no obligation on us to have an element in the language for each entry in the requirements specification. The point of the requirements specification is to define what the language must be capable of representing, not how it represents it. That comes next, when we define the <u>ontology</u> of the language.

System Dynamics modelling (stocks, flows, intermediate variables, influences...)
Disaggregation into classes (age-class, sex-class, species-class, etc)
Disaggregation into spatial layers (soil layers, forest canopy layers)
Spatial disaggregation (grid squares, polygons, zones...)
Populations of objects, with creation/deletion of object instances
Associations between objects (e.g. farmers owning fields)
Modularity: plug-and-play, unplug-and-play
Different time steps for different parts of the model
Continuous/discrete time

Table 7.1 Part of the requirements specification for the Simile visual modelling environment

The requirements specification should also include an explicit statement of what the language will not be designed to represent - a sort of non-requirements specification. This may seem an odd thing to do: why should we deliberately design a restricted language? There are several facets to the answer to this question, but the main one is that the model-representation language is no use without tools to handle models represented in this language. The greater the scope of the language (the more modelling requirements it can represent), the greater are the demands on the tools for handling models expressed in the language.

In designing the Simile model-representation language, we decided that it should not be designed to handle several modelling paradigms. Two principal ones were:

- discrete-event modelling

- optimisation models.

Discrete-event modelling represents a particular way of handling time. In continuous-time/discrete-time modelling, simulation proceeds by dividing time into discrete intervals (time steps), each representing the same amount of time. In discrete-event modelling, events happen at arbitrary points in time. The approach is widely used in the industrial area (e.g. for queuing problems in manufacture), and also underlies the message-passing paradigm used in many agent-based systems. It is, however, still very much a minority (though increasing) approach in ecosystem modelling. Some modelling systems (such as Modelica [see URLs and Appendix 1]) do allow hybrid modelling (combining discrete-event modelling with continuous-time/discrete-time modelling), but it was decided not to design Simile's model representation language to handle both views of time, given the resources we had available for tool development and the minority status of discrete-event approaches.

The main use of optimisation approaches (such as mathematical programming - e.g. linear programming) in the ecosystem field is for economic modelling, when human socio-economic factors are included in the model. Such approaches represent a fundamentally different problem-solving approach. Instead of answering the question: "What will the system be like in the future, given its present state, various inputs, and a model of how it behaves?", it addresses the question: "What should the inputs be in order to achieve some goal?". We decided not to incorporate such a problem-solving approach in Simile's model-representation language, given its very different nature.

7.2 Designing the model-representation language: choosing an appropriate level of abstraction

If you were designing a representational language for, say, electronics, it would be fairly obvious that it should be cast at the level of electronic components (resistors, transistors, etc). These are the natural building blocks for electronic circuits. If you were designing one for, say, architecture, it may be less obvious: you could cast it in terms of architectural objects (doors, windows, etc); or in terms of geometric shapes (such as lines and rectangles). There is a decision to make about the appropriate **level of abstraction** that the language should be based upon.

The problem of choosing an appropriate level of abstraction is much greater in ecosystem modelling, for three reasons. First, ecology covers a very broad range of scales, from the molecular up to the global, with quite different natural units at each scale. Second, even at one scale, modellers adopt quite different approaches. For example, a population may be modelled: as a collection of individuals; as a single state variable, or as an age-class structure. Third, the number of terms that ecosystem modellers use in describing their models is very large (unlike, say, electronics), so there is a danger of building a language with a very large vocabulary if we simply aim to use the terms used by modellers. For example, a modeller may specify that a spatial model is based on a 'spatial grid' with '4-edge-cell neighbourhood' and 'wrap-around edges'. Should these be basic ('primitive') terms in the language?

For guidance, I propose that the following four principles (or design goals) be used to direct the design of a model-representation language for ecosystem modelling.

parsimony: the language should have a small number of terms;

expressiveness: the language should be capable of handling the wide range of modelling requirements expressed in the requirements specification (above);

intuitiveness: the terms in the language should be natural terms for a modeller to use;

extensibility: new terms can be added into the language.

Parsimony is desirable to make the language easier to learn, both for building models and for 'reading' other people's models. It also simplifies the task of developing tools for handling models. **Expressiveness** is required to be able to cope with the very wide range of models - and modelling paradigms - used in ecosystem modelling. **Intuitiveness** is desirable to minimise the conceptual gap between the way that a model is conceptualised by the model-designer, and the way it is represented on a computer. **Extensibility** reduces the need to 'get it right' from the beginning. It also allows high-level, domain-specific concepts (such as '4-edge-cell neighbourhood') to be expressed for a specific community of users, in terms of more basic language elements.

These principles sometimes pull in opposite directions. For example, the terms in a language which is both parsimonious and expressive will probably express rather abstract, low-level concepts: this mitigates against intuitiveness, since modellers prefer to think in more concrete terms. The challenge for the designer of a model-representation language is to find the right balance between the four design objectives.

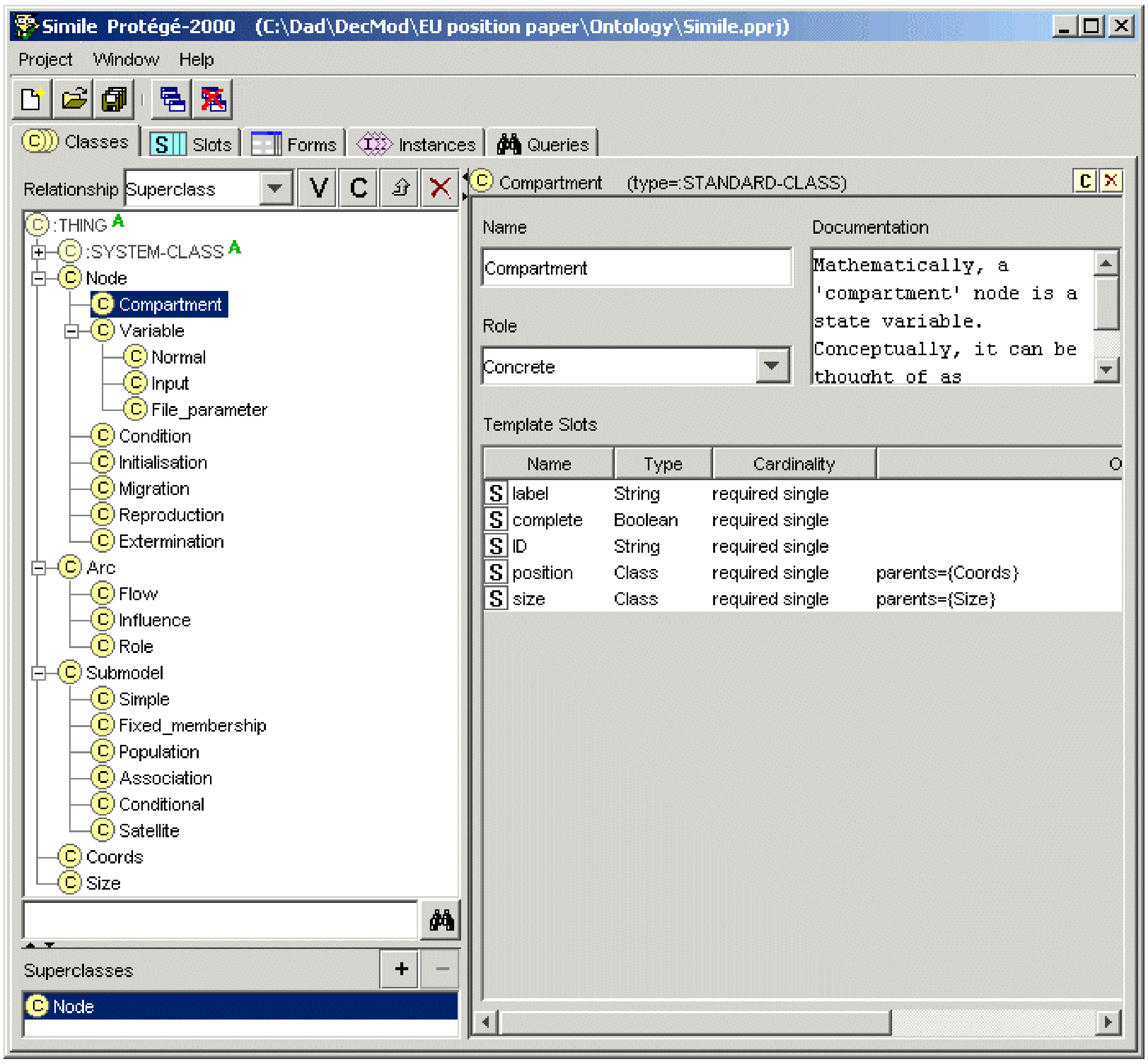
The design of the Simile model-representation language (described in Section 8) was driven by these design goals. It is cast at a rather low level of abstraction, in terms of mathematical quantities and a generalised notion of 'object'. As a result, it is capable of handling a wide variety of modelling paradigms, with some trade-off in terms of how intuitive it is to express certain modelling concepts. See Muetzelfeldt (2003) for a more detailed analysis of the issues involved in developing a unified modelling language. The design choices involved in developing Simile's language can be contrasted with those adopted by Fall and Fall (2001), in designing a domain-specific language for modelling landscape dynamics.

7.3 Designing the model-representation language: ontology

The word 'ontology' is used to refer to the terms used in a language (in particular, computer-based languages), the relationships between the terms, and their meaning **(**http://www.ontology.org).

For example, a language for representing electronic circuits might include the terms **resistor** and **passive_device**, the relationship 'resistor **is_a_type_of** passive_device', and a definition of what a resistor is. Some aspects of an ontology may be defined formally, in a form that can be represented on a computer: for example, the list of terms and possible synonyms, and the **is_a_type_of** relationships. Other aspects may be defined in a prose document: for example, the precise definition of a transistor.

The need to be precise in defining the terms used in a particular area of informatics is now widely recognised, and there are numerous ontology-development initiatives in many subject areas (http://www.ontology.org). This is seen as particularly important in permitting information exchange between software systems, especially over the internet, since commitment to a shared ontology enables two software systems to share information confident in the belief that they each share the same understanding about what the information means. This is crucial to the development of the Semantic Web, in which web-based programs exchange and process information without human intervention.


Simile Protégé-2000 (C:\Dad\DecMod\EU position paper\Ontology\Simile.pprj)
Project Window Help
Classes Slots Forms Instances Queries
Relationship Superclass

```
:THING
  :SYSTEM-CLASS
  Node
    Compartment
    Variable
      Normal
      Input
      File_parameter
    Condition
    Initialisation
    Migration
    Reproduction
    Extermination
  Arc
    Flow
    Influence
    Role
  Submodel
    Simple
    Fixed_membership
    Population
    Association
    Conditional
    Satellite
  Coords
  Size
```

Superclasses: Node

Compartment (type=:STANDARD-CLASS)

Name: Compartment

Role: Concrete

Documentation: Mathematically, a 'compartment' node is a state variable. Conceptually, it can be thought of as

Template Slots

Name	Type	Cardinality	O
label	String	required single	
complete	Boolean	required single	
ID	String	required single	
position	Class	required single	parents={Coords}
size	Class	required single	parents={Size}

Fig. 7.1 A partial view of Simile's ontology, in the Protegé ontology-design software

Figure 7.1 is a partial view of the ontology underlying Simile's model-representation language. This is presented in the form of a screen dump from Protegé, one of a number of software packages specifically developed for ontology design. It shows, on the left, the taxonomy of model-building elements: for example, it shows that **compartment** is a type of **node**. On the right, we see more information for the selected class, including its 'slots' (attributes) and some free-form text. We see that **compartment** has the attributes **label**, **complete**, **ID**, **position** and **size**. **label** and **ID** are string (character) attributes. **complete** (denoting whether the element has the required mathematical information associated with it) is a Boolean (true/false) attribute. **position** and **size** are defined by another class, which in turn represent the fact that each has two numeric values. Each slot is required to have a single value.

We can evaluate Simile against the design criteria. Its model-representation language is very expressive: a wide range of ecological and environmental simulation models can be represented in it (Muetzelfeldt 2003). It is parsimonious, having a language based on only some 12 symbols. And it is pretty intuitive, using the widely-used System Dynamics notation plus a clear symbolism for representing objects.

7.4 Designing the model-representation language: syntax

The ontology defines what we might consider to be the abstract language: the set of concepts that our language is capable of representing. In order for models to be actually represented on a computer, we require a syntax for statements in the language. This might be in binary form - written and read only by a computer program - or it could be in text form, so that it can be processed both by computers and also by humans (entered using a word processor, and printed out).

A few years ago, a group of people designing a new language might very well have come up with a new syntax, without regard to the syntax of languages designed for other purposes. For example, the statement

```
compartment: name=biomass, initial=100
```

might represent the fact that the model contains a compartment 'biomass' with an initial value of 100.

Over the last few years, however, there has been a huge increase in the use of **XML** (the eXtensible Markup Language) as a **metalanguage** within which other languages can be expressed. Section 6.7 introduces XML notation for representing model structure, and Appendix A2 provides some more detail. I simply reiterate here that XML is the obvious metalanguage to use for representing simulation models. Not only does it give access to the technologies developed around it, but it also means that some parts of the notation exist already, such as the use of MathML [see URLs] for representing equations.

There is not, however, a one-to-one correspondence between the ontology for a language and its syntax, even given a commitment to the XML metalanguage. There are still many choices to be made about representation, which will affect the readability of model description files (if that is desired), the size of the files, and the ease of processing by software tools.

XML supports the concept of **namespaces**. In this context, this enables all the language elements that relate to a particular modelling paradigm - such as System Dynamics - to be grouped together. This means that groups developing software tools do not have to commit themselves to dealing with all concepts that the full language supports: rather, one tool might restrict itself to (for example) the System Dynamics namespace.

7.5 Software tools for processing models

With conventional modelling, the program used to implement the model is at the same time the tool used to do one particular task with the model: simulate its behaviour.

With declarative modelling, we separate the representation of the model from the tools available to process it in some way. One such tool can be a simulator; but there are many other useful ways of processing a model apart from simulating its behaviour. The same tool can be re-used for any model expressed in our model-representation language. Conversely, the same model can be processed by a wide variety of tools, depending on what we want to do with the model.

What do is meant by 'processing the model'? Well, one major aspect is to produce a runnable version of the model: this enables us to simulate the behaviour of the model. This is what we normally think of doing with a model and, because it is so important, it will be discussed in more detail below, in Section 7.6. Other forms of processing are considered here:

Designing and editing a model

When a model is considered as a declaratively-represented design, we can envisage software tools to support the process of designing and editing the model - a computer-aided design (CAD) package for ecosystem modelling. There are a number of visual modelling environments that support System Dynamics modelling, such as Stella, Vensim, Modelmaker and Powersim. Simile is another example, and is described in Section 8. In Simile's case, the resulting model can be saved in an open, text-based model-representation language, and thus available for processing by other, independently-developed tools.

Displaying model structure

Tools can be developed for displaying model structure in a variety of ways. Section 8.3 gives examples of tools for displaying any Simile model as HTML (a web page, viewable in a web browser, with hyperlinks between different parts of the model), as a diagram viewable in a web browser, and even as spoken text, for people with impaired vision.

Comparing two models

Consider the case of groups within a research programme developing a joint model, then work on it independently for 6 months. They then want to compare the two versions, with a view to possibly merging them or discussing differences. If the model is a program, then at best they can use text-comparison ('diff') software. It would be far better if the actual design of the models could be compared, and this is only possible if the models are represented declaratively.

Transforming one model into another

The declarative representation of a model consists of a set of symbols. These symbols can be manipulated by an appropriate computer program to produce another set of symbols, which can represent the design for a different model. In other words, it is possible to transform one model into another. Muetzelfeldt and Yanai (1996) introduce this concept in the field of ecological modelling.

There are two main uses for this in ecosystem modelling. The first is to explore alternative designs automatically. Any model embodies a large number of design choices, many of which could well have been different if the model had been developed by a different but equally-skilled researcher. Do these differences matter? The only way is to build a variety of models and try them out. Conventionally, this would require hand-coding the changes into the model program (and thus is rarely done), but a model-transformation tool, drawing on a body of knowledge about legitimate alternative designs, could do this automatically.

The second is in scaling, and simplifying complex models. There is frequently a need to simplify complex models in order to make them computationally tractable, and to enable general principles of their behaviour to be analysed (see Dieckmann *et al* 2000 for a detailed study of this in the context of spatial modelling). The Earth System modelling community is particularly concerned with this, since much of the understanding of processes - and much of the modelling - is at a fine scale (e.g. carbon dynamics of a forest), while answers are needed at a global scale. Yet we cannot simply replicate the fine-scale models across a global grid, since the computational cost would be too high. There is thus a need to have a variety of models, from extremely detailed global models, through Earth Models of Intermediate Complexity (EMICs), to very simple 'Daisyworld' models. Currently, each model is developed separately: model transformation offers the prospect of doing this automatically, using expert knowledge on how to collapse detailed models into simpler, more aggregated versions.

Interrogating model structure

When a model is a declaratively-represented design, it is possible to develop tools for asking questions about the design. For example, one could ask about all the variables in a model that are influenced by the variable 'temperature' directly, or indirectly over 3 influence links. Or one could ask to see all compartments that had just two outflows.

Finding models in a model catalogue

Extending the idea, one could have tools for interrogating a catalogue of models (which could be held in one place or dispersed over the internet), in order to find all models with certain properties. For example, one could ask for all models which had a photosynthesis flow influenced by temperature and sunlight. Model catalogues exist already, notably REM, the Register of Ecological Models at the University of Kassel, Germany [See URLs]. However, these catalogue models by separately-written metadata. With declaratively-represented models, tools could search according to the actual structure of the models themselves, reducing the effort needed for metadata preparation, guaranteeing consistency with the actual model, and providing access to every detail of the model.

This raises significant issues about common standards for naming ecological entities. Initially, such tools could work much as Google, looking for exact text matches. Eventually, a common vocabulary (ecological ontology) would be developed, rather along the lines of the CAB Thesaurus developed by CAB International, or the AgroVoc Thesaurus developed by FAO [See URLs].

7.6 Producing a runnable version of the model

How do we simulate the behaviour of a declaratively-represented model? This is a key question: after all, that's why most models are made (and that is really all that is done at the moment with most models).

We have two main options. We can either write an interpreter for the model that operates directly on a declarative representation. Or we can generate a program in a conventional programming language from the declarative representation, and run that program.

Simulation using an interpreter

Writing a program that can solve equations given to it as data is technically fairly straightforward. Consider the following simple model:

dA/dt = 2.5 - 0.1*A

This describes the dynamics of a single compartment, with a constant inflow and a flow out at a rate proportional to the compartment's contents. We can re-express the right-hand side in Reverse Polish Notation, as follows:

dA/dt =	**2.5**	**0.1**	**A**	*****	**-**

The 5 elements (values, variable and operators) can be stored in 5 elements of an array, which can then be processed to calculate the result. This is then used to change the value of A, and the process repeated.

The main problem with this approach is that it tends to be computationally inefficient, since another layer of computation (processing the expression held in Reverse Polish Notation) is involved. It is therefore unsuitable for complex models.

Generating a procedural program to simulate model behaviour

Generating a conventional procedural program from a declaratively-represented model offers the prospect of having models that run fast. It is generally thought that the declarative vs procedural issue hinges on a trade-off: declarative=intuitive, procedural=efficient. By generating a program (which can then be compiled), we get the best of both worlds. In fact, possibly even better! - since the program generator can use various optimisation tricks that even quite good programmers might not incorporate.

Generating a procedural program from a declarative representation is conceptually (and technically) much easier than building an interpreter. Consider the above example. Once we have extracted the variables and the equation we can use some canned program lines to produce (bits in bold are extracted from the declarative representation) a program like the following pseudo-code:

```
input A
dt = 0.1
for time = 1 to 20
for time1 = dt to 1.0 step dt
dAdt = 2.5 - 0.1*A
A = A + dAdt*dt
next time1
print time, A
next time
```

If this were in C, then it could be compiled and would run very efficiently.

What sort of environment should be available for running simulations with a runnable version of the model? Exactly the same as for any other type of modelling framework. If a model is implemented as a stand-alone program, without working within a framework, then usually what happens is that input, analysis and display routines ("support routines") are produced, also on a one-off basis - a huge waste of effort. The adoption of any sort of modelling framework, whether it be using a component-based approach or a declarative modelling approach, necessarily introduces a separation between "the model" and the support routines needed for exploring the behaviour of the model. We will not consider this further in this document, since it is an issue not specifically related to declarative modelling. But it is certainly an advantage of declarative modelling that this separation is forced upon us, so that a toolkit of support routines can be developed, with the effort involved in producing them amortised over many models.

7.7 Modelling environments

One of the strong benefits of the declarative modelling approach is that it encourages the development of discrete tools for processing models. A group does not need to undertake the development of a large and complex software package in order to produce something of use to the rest of the community.

Nevertheless, there is still a place for the software packages (modelling environments) that combine a number of tools under one roof. In some cases, these will be simply tools that could be used independently packaged together: they will still communicate the model structure using files in the XML-based modelling language. In other case, the tools might be more closely coupled, sharing a representation of the model structure held in memory. The key need is to ensure that the various packages adhere to a common external standard for sharing models, rather than engaging in 'feature creep' to try to get an edge over rivals and tying users into their own proprietary system.

One concept that will almost certainly emerge is that of **workflow**. Some tools produce a model as output, others input a model, while others do both. We can therefore envisage a chain of operations, constituting a higher-level task. For example, a user might wish to routinely compare two models (perhaps the latest versions from two collaborating institutes), produce a new model combining the recent changes, transform this into a simpler model, then display this as an HTML page and run it under standard conditions. Workflow languages and design systems exist, and they can be applied to a sequence of modelling tasks. For example, WSFL (the Web Services Flow Language) [see URLs] has been designed for doing just this task for web services.

7.8 Publishing the models

I have suggested that XML be used as the metalanguage for representing models. Now, XML is intended for distributing machine-readable content over the web, just as HTML is intended for distributing human-readable documents. It therefore follows that we can 'publish' models represented in XML, just as we can publish web pages.

We can therefore envisage that, in the future, you will be able to search for models just as you now search for web pages, perhaps using a search engine like an XML-enabled Google. Clicking on the model could automatically load it into your favourite modelling environment, or you could load a model in your modelling package by entering its URL rather than a local file name. Specialist web crawlers could build up model catalogues automatically by searching out models, and some people would build up their own catalogues manually just as people currently construct lists of web sites. Building a model could involve selecting submodels published in a variety of countries with a few mouse clicks.

The published models can be viewed directly in a standard browser - both Netscape and Internet Explorer are able to display XML using a Windows Explorer-type interface. This is unlikely to be particularly digestible for the average user, since it will contain much technical information on the model structure. More usefully, the time will come shortly when XML can be viewed through the filter of an XSLT document (the eXtensible Stylesheet Language for Transformations - see Appendix A2.2), enabling it to be simplified and displayed as HTML or diagrammatically in your browser, or even converted into an audio presentation (Section 8.3).

7.9 Conclusions

The architecture for a declarative modelling approach is unusual, in that the emphasis is placed firmly on the specification of a model-representation language, rather than on the design of a software package. The design of the language is a challenging task, requiring a balance to be struck between a number of competing objectives. Once it has been specified, tools for supporting the modelling process can be developed in a piecemeal fashion, to satisfy a wide variety of modelling requirements.

The next Section illustrates the general ideas presented here, using the Simile visual modelling environment as a proof-of-concept demonstrator.

8. Proof-of-concept: the Simile visual modelling environment

Simile is a visual modelling environment specifically developed to explore and demonstrate the feasibility and benefits of a declarative modelling approach in ecological and environmental research. It is now a powerful modelling environment used in a number of international research projects (Muetzelfeldt and Massheder, 2003).

Simile is presented here as a 'proof-of-concept demonstrator' for declarative modelling. It has been chosen for two main reasons. First, its model-representation language is rather more expressive than that of other modelling environments, and thus constitutes a more convincing exemplar. Second, Simile was developed specifically to make the case for declarative modelling, so it brings these concepts to the surface to a greater extent than other environments.

There is no claim here that Simile is all that is needed for a declarative modelling approach in ecosystem research. It has been developed on modest resources, and without community-wide consultation. Its aim is to stimulate the community to develop an infrastructure for systems modelling, by showing what's possible and exploring some aspects of a declarative modelling approach.

8.1 Simile features

Simile is characterised by the following main features:

<u>System Dynamics modelling</u>
System Dynamics is a diagrammatic notation for representing systems of DAEs (differential-algebraic equations), based on a concept of compartments (stocks, levels) and flows (processes) (Forrester 1971). The compartments represent mathematical state variables, and the flows contribute to the rate-of-change term for each state variable. Additionally, System Dynamics includes the concept of influence, intermediate variable and parameter. Influences arrows show which quantities are used to calculate which other quantities. Intermediate variables hold intermediate quantities. Parameters hold values which are constant during a simulation run.

<u>Disaggregation</u>
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.

<u>Object-based modelling</u>
Simile allows a population of objects to be modelled. As with disaggregation, you define how one member behaves, then specify that there are many such members. In this case, however, the model designer can add in symbols denoting the rules for creating new members of the population, and killing off existing members. Individual members of the population can interact with others.

<u>Spatial modelling</u>
Spatial modelling, in Simile, is simply one 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 interactions between spatial units can be represented (Muetzelfeldt and Duckham, in press).

<u>Modular modelling</u>
Simile allows any model to be inserted as a submodel into another 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 the submodels of a complex model.

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

<u>Customisable output displays and input tools</u>
Simile users can design and implement their own input/output procedures, independently of the Simile developers. Thus, users can develop displays for model output that are specific to their requirements. Once developed, these can be shared with others in the research community.

<u>Declarative representation of model structure</u>
A Simile model is saved in an open format as a text file (in Prolog syntax). This means that anyone can develop tools for processing Simile models in novel ways. 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 efficient of models across the internet (as XML files), and for the sharing of models between different modelling environments.

8.2 Simile in action

In this section, I present a simple model, give examples of the types of output that the model produces, and illustrate other tools that can be used for handling the model.

The model is a simple crop growth model, of the type that might be used to explore the effect of climate change on the provision of ecosystem services. The model is a deliberately highly-simplified one, prepared for this paper to show the types of model that Simile can handle and what a model looks like in Simile.

<u>The model diagram</u>

Fig. 8.1 shows the model diagram in Simile for a spatial crop model.

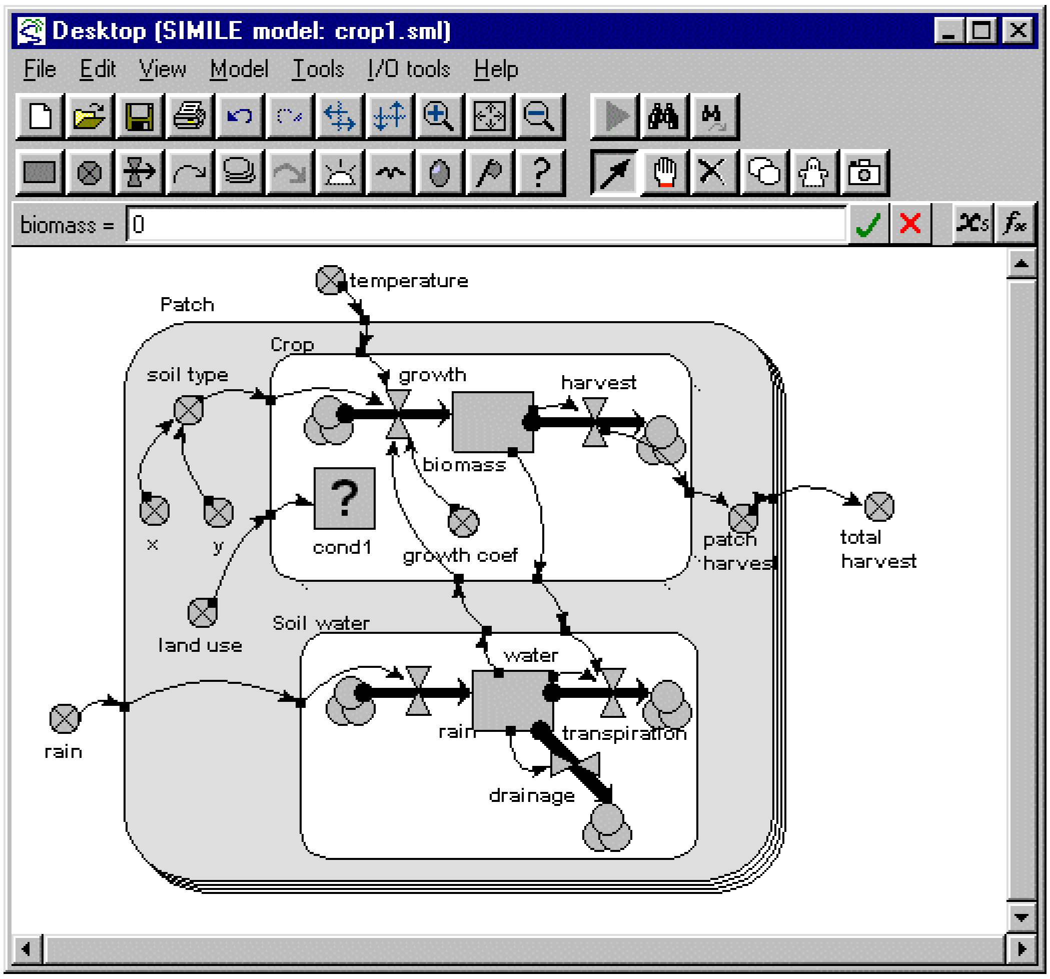


Fig. 8.1 A spatial crop model in Simile.

The model represents an area of 40x40 one-hectare patches. This is represented by the multiple-instance Patch submodel. We can see from the model diagram that there are multiple instances of the Patch submodel - the information on the number of patches is part of the mathematical specification of the model. The patch submodel contains a crop submodel and a soil water submodel. Each patch simulates its own soil water dynamics. Each patch may simulate crop dynamics in it, depending on whether it actually contains a crop (this depends on the land use of that patch): note the condition symbol (the question-mark) inside the crop submodel. Crop growth is influenced by soil water, soil type and temperature. Loss of water from the soil is influenced by crop biomass. Crop harvest is totalled once a year across all 1600 patches.

The model equations

In Simile, as with other visual modelling environments, modelling is a two-level process. The model diagram is developed at the upper level, while the mathematical part of the model - values and equations - are entered at the lower level (by opening up a dialogue box for each model element).

Table 8.1 lists all the equations in the spatial crop model. The text is almost in the form produced by Simile's mechanism for exporting model equations (including the comments, which can be added to any model element). The only difference is that it has been formatted here for improved presentation.

Desktop	
temperature = graph(time(1)) + time(1)/20	*Temperature follows an annual cycle, expressed in a sketched graph, plus a rise of 1 degree C every 20 years. (Simile provides a tool to enable users to sketch the graph, and uses linear interpolation between the points to evaluate the sketched function.)*
total harvest = sum([patch_harvest])	*We sum the harvests from all the patches to give the total for the whole system.*
rain = rand_var(0,2500)	*Rainfall varies randomly on each time step, with an annual average of 1250.*

Submodel Patch	
land use = floor(rand_const(0,2))	*This determines randomly the land use (i.e. whether a crop is grown) for each patch.*
x = floor((index(1)-1)/40)+1 **y** = fmod(index(1)-1,40)+1	*These two expressions give each patch unique x,y coordinates on 1 range of 1 to 40.*
soil type = (if x+y>rand_const(45,55) then floor(rand_const(1,1.3)) else floor(rand_const(1.8,3)))	*As a demonstration, the soil type is determined partly randomly, partly in terms of the x and y coordinates of each patch.*
patch harvest = sum({harvest})*dt(1)	*This works out the amount harvested at the start of each year for each patch.*

Submodel Patch/Crop	
cond1 = (land_use==1)	*This condition determines which patch is grown, determined by the patch's land use.*
biomass = 0	*Initial crop biomass is set to zero.*
growth = (if soil_type==1 and water>100 then 20*0.04*temperature elseif water>150 then 18*0.04*temperature else 0)	*Crop growth is worked out from soil type, soil water content and temperature.*
growth coef = 17.0	*The growth coefficient (the growth rate for optimal conditions of temperature and soil water) is different for each crop, being 17,20 and 24 tonnes per year for crop types 1, 2 and 3 respectively.*
harvest = (if fmod(time(1),1)==0 then biomass/dt(1)else 0)	*Current crop biomass becomes the harvest flow at the start of each year.*

Submodel Patch/Soil water	
water = 200	*Initial soil water is set to 200 mm*
rain = rain	*The rain flow for each patch is set equal to the overall rainfall.*
transpiration = (if water>50 then 20*sum({biomass})else 0)	*Transpiration depends on crop biomass, if there is sufficient water in the soil.*
drainage = 7*water	*Soil water drainage is proportional to current soil water content.*

Table 8.1 Equations for the spatial crop model.

Running the model
Fig. 8.2 shows a graph for total harvest over 50 years, for two runs: no change in temperature (lower line), and a 2.5 degree change in temperature (upper line).

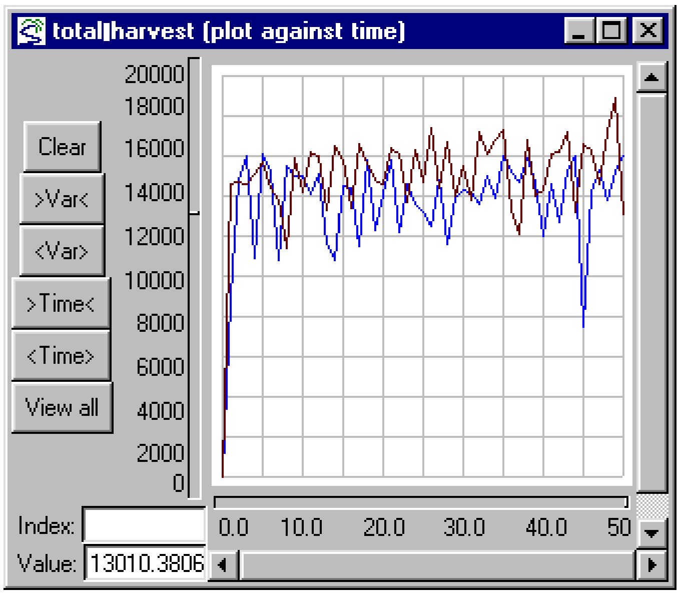


Fig. 8.2 Graph showing total harvest for two separate runs of the spatial crop model.

Since the model is a spatial one, it is also possible to view the results at some point in time as a map. **Fig. 8.3** shows maps for annual harvest and soil water at the end of the year.

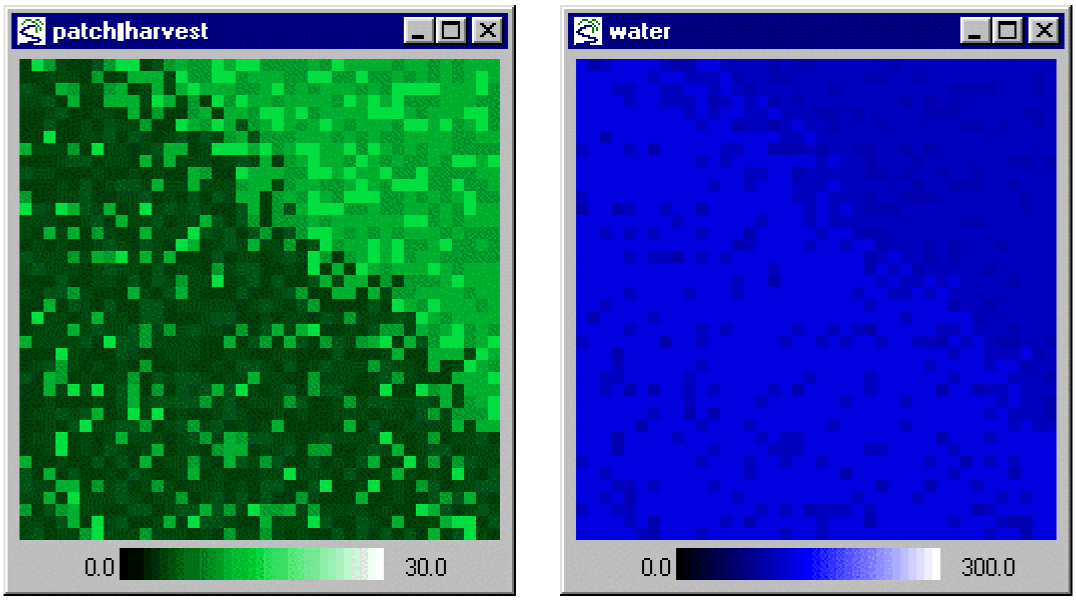


Fig. 8.3 Maps showing crop annual harvest and soil water at the end of a year.

8.3 Exploiting the declarative nature of Simile modelling

When you are using Simile, you build the sort of model presented above. Clearly, while you are doing this, Simile maintains some sort of internal representation of what you do: where symbols appear on the screen, the names you give them, the equations you type in, and so on. This is necessarily declarative: it is a set of facts describing the current state of the model. Simile only generates a procedural version of the model - a computer program - when you decide that you want to run the model.

What happens when you decide to go off and have lunch? You save the model in a file, and close Simile. This file is then loaded again when you come back from lunch, and you see the model diagram with all the information you were working on before. The file contains the same information as the internal representation. This is thus also a declarative representation of the model structure, but with one important difference: it is an external representation of the model, one that in principle can be processed by other software tools, not just Simile.

All the above is true, in principle, for other modelling environments, such as Stella and Modelmaker. However, there is one critical difference. In general, the other software packages use a proprietary binary format for saving their models.[1] Simile, in contrast, uses an open text format: anyone else is free to understand this format and write programs that are capable of processing the files. In principle, the same could be done for models represented in a binary format. In practice, since the format is not published, it is difficult or impossible for third parties to develop appropriate software.

Since this section is about Simile as a proof-of-concept for declarative modelling, I will now demonstrate the feasibility of the claim that tools for processing Simile models can be developed independently of Simile itself. I will give 3 examples:

- generating HTML descriptions of any Simile model;

- marking-up the model for an audio-generation system;

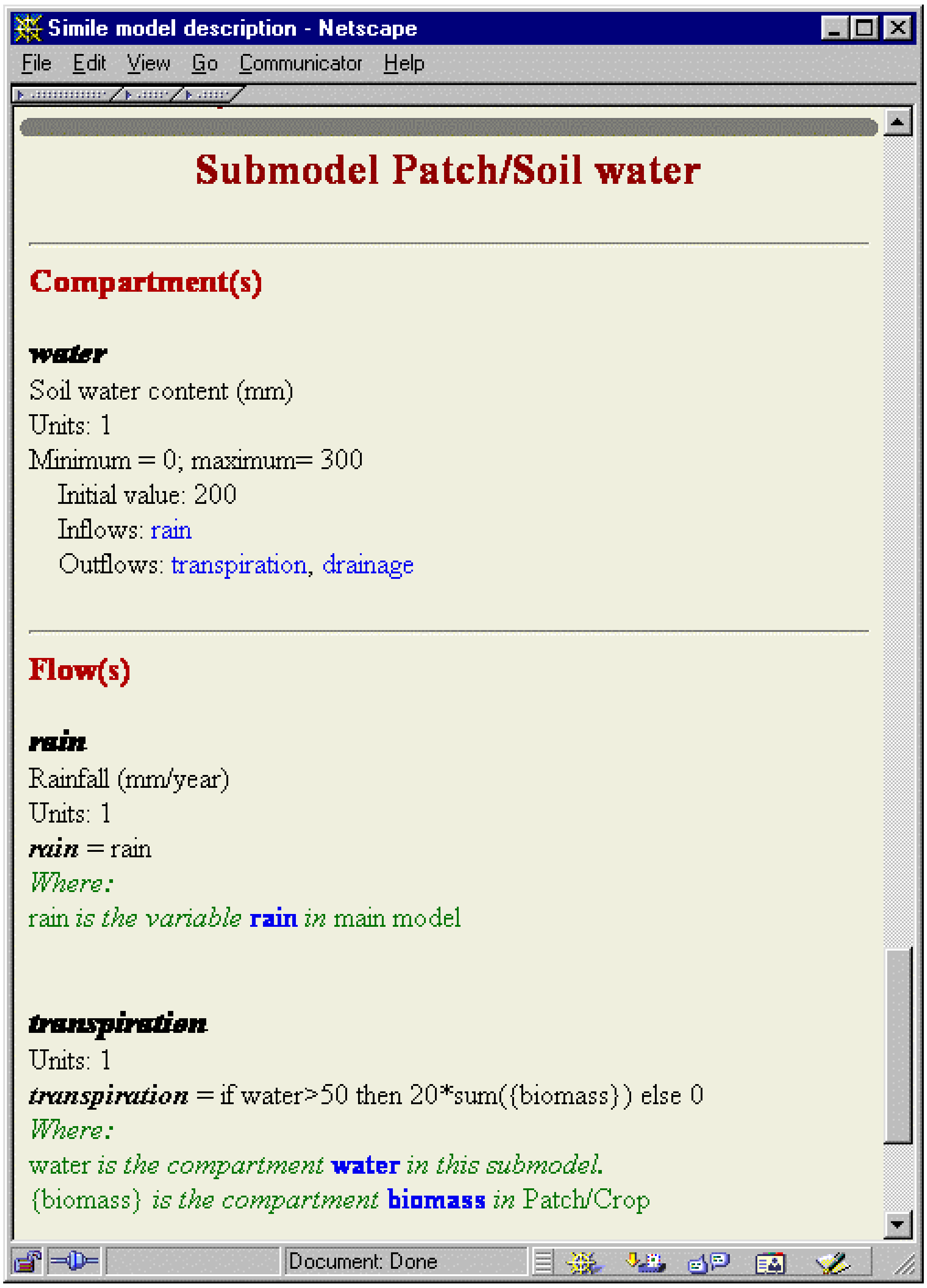
- generating a diagrammatic rendering of the model for viewing in a web browser..

[1] Some packages, such as Stella, do enable you to obtain the model equations in text form, but the format is not designed for processing by other software tools, and the diagrammatic aspects of the model are held in the binary saved-model file. Vensim does make the complete model description, including the diagrammatic aspects, available in text form, but this is not in a standard markup language.

Example 1: The HTML generator for Simile models

This is a Prolog program that generates a text document containing HTML markup for any Simile model. The document is organised submodel-by-submodel. For each submodel, it displays the model elements according to category: compartments, flows, variables, etc. For each element, it displays any information it has on it, including comments, min/max values etc, with nice formatting. An extract from the HTML generated for the model above is shown in the Fig. 8.4.

This tool also includes hyperlinks to enable browsing around the model. You can click on any variable used in an equation to be taken directly to where that variable is defined. You can thus very quickly chain through a whole sequence of variables, just using the mouse.


Simile model description - Netscape
File Edit View Go Communicator Help
Document: Done

Submodel Patch/Soil water

Compartment(s)

water
Soil water content (mm)
Units: 1
Minimum = 0; maximum= 300
Initial value: 200
Inflows: rain
Outflows: transpiration, drainage

Flow(s)

rain
Rainfall (mm/year)
Units: 1
rain = rain
Where:
rain *is the variable* **rain** *in* main model

transpiration
Units: 1
transpiration = if water>50 then 20*sum({biomass}) else 0
Where:
water *is the compartment* **water** *in this submodel.*
{biomass} *is the compartment* **biomass** *in* Patch/Crop

Fig. 8.4 A portion of the HTML generated for the spatial crop model, viewed in Netscape.

Example 2: Generating audio descriptions of model structure using VoiceXML

This example shows the potential for presenting models in novel ways, once the step has been taken to represent models in XML.

As mentioned previously, XML is the basis for a very large number of markup languages (see http://www.coverpages.org/xmlApplications.html). One of these, VoiceXML, is for marking up text so that it can be delivered by telephone, though the sort of automated answering systems that irritate us when we are trying to talk to a real person in some organisation. However, for people with impaired vision, this technology has huge potential, whether delivered by phone or through a computer loudspeaker.

The following text is an actual VoiceXML document, with some extraneous bits removed and formatted to make its structure clearer. The bits in bold are text that is spoken by the computer-synthesised voice: the underlined parts are generated for a specific model. When users phone through to a voice portal service, they will be asked to say one of the words "top", "patch", "crop" or "water", and you will then get a brief description of the relevant part of the model.

```
<?xml version="1.0"?>
<vxml version="1.0">

<form id="main">
<audio>    Welcome to the Simile VoiceXML demonstration.
           Description of the spatial crop model.
           The model contains 4 submodels: top, patch, crop, water,
</audio>

<prompt>   Please say: top, patch, crop, water,
</prompt>

<audio>    You have selected <value expr="action"/>
</audio>

<if cond="action == 'top'"> <goto next="#top"/>
    <elseif cond="action == 'patch'"/> <goto next="#patch"/>
    <elseif cond="action == 'crop'"/> <goto next="#crop"/>
    <elseif cond="action == 'water'"/> <goto next="#water"/>
</if>
</form>

<form id="top">
<audio>    The submodel top contains the 3 variables temperature, total
           harvest, rain,
</audio>
<goto next="#main"/>
</form>
</audio>
<goto next="#main"/>
</form>

<form id="water">
<audio>    The submodel water contains
           the 1 compartments water ,
           the 3 variables var9, biomass, rain,
           the 3 flows rain, transpiration, drainage,
</audio>
<goto next="#main"/>
</form>

</vxml>
```

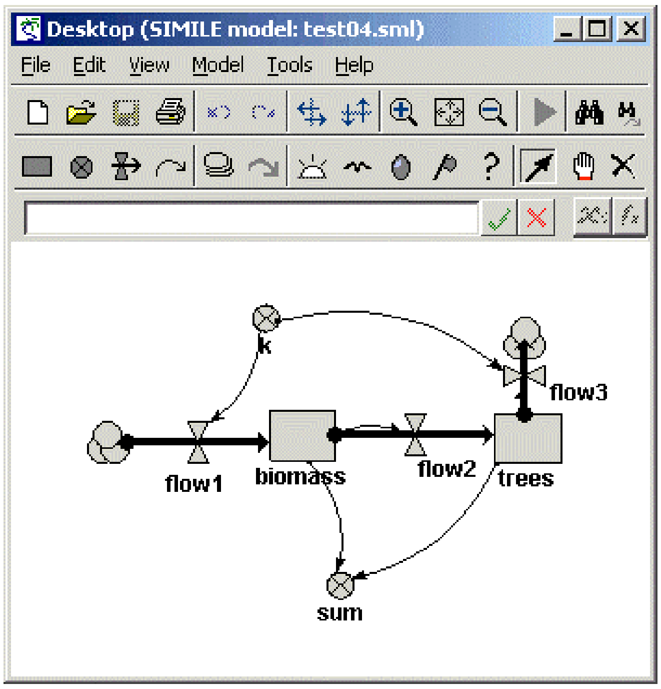
Example 3: Generating graphical views of model structure, viewable in a web browser

Simile, and similar software, have shown clearly the value of diagrammatic representations of model structure for building and communicating models. However, to date such tools operate as stand-alone software products on one's own PC. New modelling will involve people searching for models or submodels across the web, and it's important that we develop the most useful ways of informing these people about the various models they encounter. Along with prose descriptions, HTML and other text-based presentations, it is desirable to use visual methods as well.

As it happens, until recently there have been no single standard for displaying vector graphics (lines, rectangles, circles etc) in web browsers. (There were a number of approaches under development, such as PGML - Precision Graphics Markup Language - and VML - vector Markup Language - but none of these satisfied all the requirements of a vector-graphics language for web browsers.) Consequently the W3C (the World Wide Web Consortium) set up a working committee to come up with a standard language, and the result is SVG (Scalable Vector Graphics) [See URLs]. Being XML-based, it can be incorporated into XML documents, and is a sophisticated graphics language in its own right, with the ability to support (e.g.) user interaction and animation.

I have developed an XSLT program to illustrate the ability to display any Simile model diagram in SVG, and thus to enable it to be viewed in a web browser. Please note that this is a very early prototype.

This file can be viewed in any web browser (Netscape, Internet Explorer) that has a SVG plug-in (obtainable from http://www.adobe.com/svg), as shown in **Fig. 8.5**. In the near future, browsers will have an SVG capability as standard.


Desktop (SIMILE model: test04.sml)
File Edit View Model Tools Help
k
flow3
flow1
biomass
flow2
trees
sum

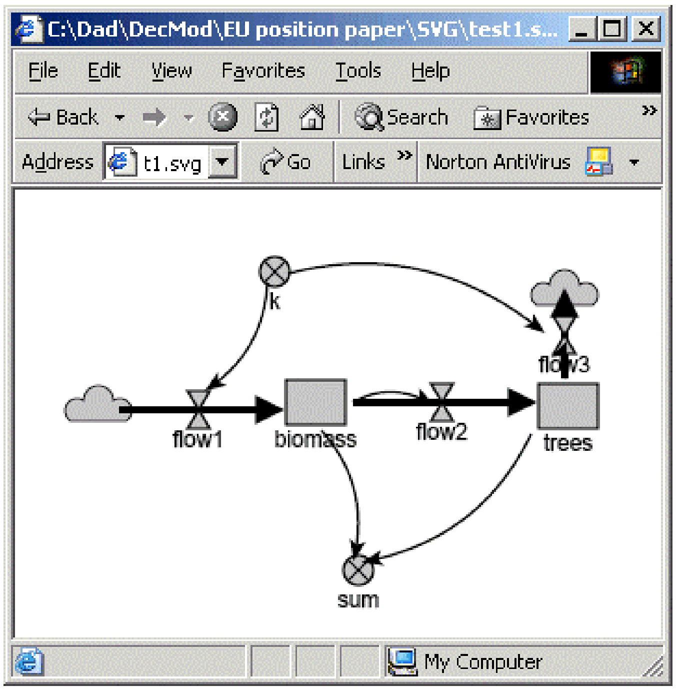

C:\Dad\DecMod\EU position paper\SVG\test1.s...
File Edit View Favorites Tools Help
Back Search Favorites
Address t1.svg Go Links Norton AntiVirus
k
flow3
flow1
biomass
flow2
trees
sum
My Computer

Fig. 8.5 A Simile model, shown in Simile itself (left), and rendered in a web browser (Internet Explorer) (right)

8.4 Conclusions

The aims of this section have been to demonstrate the feasibility of a declarative modelling approach, and that its theoretical benefits are realisable in practice.

Quite possibly, some modelling environment other than Simile could have been used as a proof-of-concept demonstrator. However, there are several reasons for using Simile here: its modelling language has considerably greater expressiveness than others; it is capable of generating efficient simulations for complex models; and it was - unusually - developed specifically on an agenda of an open, shareable modelling language.

Nevertheless, we should see Simile merely as an indicator of what is possible, not as an indicator of the maximum that can be achieved. It has been developed on a budget that is small compared to many models, let alone other modelling environments; it has weaknesses, for example in the time required to generate the program code for large models; its use of XML as a model-representation formalism is still in its infancy; and the number of independent tools for handling Simile models is still very limited. So this is in no way a "use Simile" promotion: rather, it is to convince that even quite modest expenditure on developing a declarative modelling infrastructure for ecosystem modelling will produce great dividends.

In the next Section, we will explore what these dividends could be in a representative ecosystem research project.

9. Case study: potential application of a declarative modelling approach in an ecosystem research project

The aim of this section is to explore what ecosystem modelling would be like if declarative modelling were the standard, routine way of doing things. In order to make it more concrete, I will base the analysis on a specific European project: ATEAM (Advanced Terrestrial Ecosystem Analysis and Modelling) [See URLs]. I will first outline the project itself, then paint a picture of what it would be like if the project were undertaken using a declarative modelling methodology.

*Please note that this is a **thought experiment**. The actual ATEAM project uses models implemented as conventional computer programs. Some of what is presented here is feasible using existing Simile capabilities (see previous Section), but would require that ATEAM models had been implemented in Simile. Other aspects would require yet-to-be-developed tools. However, everything presented here is totally feasible using current technologies.*

9.1 Overview of the ATEAM project

The ATEAM project [see URLs] is an EU-funded project, started in 2001, which aims to assess the vulnerability of sectors relying on ecosystem services to global change. Ecosystem services are "the conditions and processes through which ecosystems, and the organisms that make them up, sustain and fulfil human life." It is based on the recognition that many economic sectors of human society (such as agriculture, forestry and water) are closely dependent on the functioning of underlying ecosystems, either directly or (as in the maintenance of biodiversity) indirectly.

The choice of sectors is shown in **Table 9.1**, along with the key ecosystem services that underpin them. It is clear that there is considerable overlap between these (e.g. between 'Forestry' and 'Carbon storage and energy'), and that they are not orthogonal: 'Mountains' constitute a region rather than a sector like agriculture.

ATEAM is representative of a significant trend in recent years in the commissioning of ecological and environmental research. For many years, such research was largely scientific in nature, and any links to human policy agendas were either absent or token. Now, there is a greater realisation that scientific research must be related to issues of public policy (and that issues of public policy must be underpinned by good science). This is reflected in the goals of the research projects.

It also reflects itself in a recognition that the science and the consequent interpretation of this for policy must involve stakeholders, where stakeholders are defined as *"People and organisations who have an interest in information on ecosystem services and their vulnerability to global change. Relevant stakeholders for ATEAM include natural resource managers, planners and decision-makers, both within the private and the public sector."* Each of the groups involved in ATEAM has made a considerable effort to involve stakeholders, particularly in relation to the choice of suitable indicators of ecosystem services (**Table 9.1**).

The ATEAM project is based on the use of computer models of the ecosystems underpinning several key sectors to explore various projected scenarios of global change. It has the following specific objectives:

- "to develop a comprehensive modelling framework for predicting the dynamics of services provided by major European terrestrial ecosystems at a regional scale;
- "to develop a range of scenarios for socio-economic development, land-use change, pollution levels, atmospheric composition and climate change up to the year 2100;
- "to maintain a continuous dialogue with stakeholders to ensure the applicability of results for the management of natural resources;
- "to develop a series of maps depicting regions and sectors that are especially vulnerable to global change."

Sector	**Ecosystem services**	**Indicators**	**Models**
Agriculture	Food & fibre production Power generation (biofuels, e.g. rape seed, sugar beet) Landscape diversity	Crop yield, yield variability & profitability Physical suitability of crops Management change Soil fertility, erosion & salinisation Biofuel crops suitability and yield	MAGEC (SWAP version) SUNDIAL ROTHC
Forestry	Wood production Power generation (biofuel: wood biomass) Landscape diversity	Tree productivity: wood biomass & increment (heartwood, sapwood) Felling potential, age classes & natural tree mortality Suitability of tree species Biofuel wood suitability and yield Forest management change Pest susceptibility	GOTILWA EFISCEN FORGRO-HYDRALL.
Carbon storage and energy	Carbon sequestration Power generation (biofuels)	Carbon storage in vegetation Carbon storage in soil	LPJ-DGVM STOMATE Sheffield-DGVM
Water	Water supply (households, agriculture) Drought & flood prevention Transport Power generation	Stream flow quality Stream flow quantity	Mac-pdm
Biodiversity and nature conservation	Beauty Life support processes (e.g. pollination) Future options (e.g. genetic diversity)	Species richness Species representation probability in nature reserve planning Species persistence Habitat richness	Statistical & empirical modelling
Mountains	Tourism (e.g. winter sports) Recreation (e.g. hunting, walking) Carbon sequestration Water supply	Snow (cover time, amount, elevation of snow line) Slope stability (risk of landslide) Landscape partitioning (e.g. forest/meadow) Carbon storage Water (quantities, peak flows, duration of drought periods)	RHESSys FORCLIM

Table 9.1 Sectors, indicators and models in the ATEAM project (table compiled by Schröter and Metzger, ATEAM project)

The modelling approach is based on the principle of using (perhaps with some adaptation) existing models (**Table 9.1**), rather than developing new models from scratch. This reflects the view that considerable effort has already been devoted to developing relevant models of cropping systems, forest dynamics, and hydrological processes, for example, and that there is no justification in needlessly reproducing this work. The "comprehensive modelling framework" largely consists of ensuring that the various models can accept input data defining various future scenarios (for climate, land use, N deposition and atmospheric CO_2) on a standard grid covering the whole of Europe.

The models are used independently of each other. For example, projections from the hydrology model, on water storage and runoff on a grid-square basis, are not fed into the agriculture or forestry models, even though in reality there will be links between these subsystems. The decision to not attempt to integrate the various models recognises the huge effort that would have been required to achieve this integration, given that each of the base models is a stand-alone program, implemented in possibly a different programming language. (As an aside: This is precisely why many groups undertaking landscape-level modelling have put so much effort into developing component-based modelling frameworks, in order to allow subsystem modules to be integrated into a comprehensive model.)

Considerable emphasis is place on benchmarking and validation of the chosen models in a pan-European context. Because all the models are implemented in different ways, these activities involve tools and procedures specific to each model. As we shall see, one of the benefits of a unified approach to modelling (such as declarative modelling) is that it avoids duplication of effort in developing tools for handling the runnable version of a suite of models, since all the models can be processed by all tools.

It is worth noting how, in ATEAM, the models are used to explore future scenarios. For all the modelling work, the drivers are data sets containing values for predicted key global change variables, generated (in the case of climatic variables) from standard climate change models on a grid-square basis. There are several models of climate change that can be used, and, to avoid prejudicing the results by the choice of the 'wrong' model, several are used. Any one of these models can be run under a number of policy scenarios: 'business-as-usual', cut-backs in CO_2 emissions, etc. Moreover, running any of the ATEAM models on a Europe-wide basis, and interpreting the results, is a lengthy and involved process. The result of all of this is that only a small number (c.10-15) of climate-change-scenario/climate-model combinations can be explored. The results of these analyses will be presented through reports, workshops, CD ROM and similar channels: there is no intention to give people interested in exploring other climate change scenarios the opportunity to run the models themselves. This is partly because of the cost and complexity involved in preparing the models for use by others, and partly because ATEAM concentrates on identifying the changes that may occur, in order to enable environmental managers to consider vulnerability and adaptation options.

9.2 A declarative modelling perspective on ATEAM

What follows is a thought experiment. It aims to explore what the ATEAM project could be like if a declarative modelling approach had been the standard way of undertaking ecosystem modelling prior to the start of the ATEAM project. (The question of how this might actually come about, including the re-casting of existing models in a declarative model-representation language, will be addressed in Section 10.)

We assume the existence of the same models currently used in ATEAM. However, as part of the thought experiment, we assume that each of these models is represented in an **XML document**, conforming to an **XML Schema**. The Schema itself implements a particular model-representation **ontology**. Each XML model document is either a complete model, or contains references to other XML documents which are included in the model as submodels.

Finding models

ATEAM involved the selection, by experts, of the most suitable models from a relatively small pool of candidate models. In a declarative modelling world, we can expect the pool of available models to be rather large (since building and modifying models becomes much easier), so the process of finding and choosing the most suitable model becomes more of an issue.

The XML model documents will be available on the web. They are published in the same way as HTML pages; made available using peer-to-peer file-sharing services (such as Gnutella); or held in a database and accessed via a web service.

Suitable models can be discovered in one of several ways. One can go to a web site that has built up a list of models, with hyperlinks. Or one could envisage an XML-enabled Google-style search engine. Or one could use UDDI (Universal Description, Discovery and Integration) [see URLs] - originally intended for finding businesses over the internet, but now being used for discovering web resources themselves.

Viewing models

How do you decide whether a particular model is suitable? You can, of course, read documentation about the model. This is your only option for models implemented through programming, but with an XML-represented model you can view the model or interrogate its structure. An exciting possibility is to view the model in a standard web browser through an XSLT filter, so that you can study it in whatever form is most useful to you. It's as though you could put on magic glasses to look at the model:

- If you put on the HTML glasses, you see the model described in an HTML page, with hyperlinks between the variables in an equation and the place where it is defined;
- If you put on the SVG (Scalable Vector Graphics) glasses, you see the model appear as a diagram (rather like the one you see in Simile);
- If you put on the VoiceXML glasses (or rather, headset!), you get a spoken description of the model;
- If you put on the Prose glasses, you get a prose description of the model.

In all cases, the presentation of the model is generated on the fly.

Loading the model

Once discovered, a single click on the link to a model downloads it into a modelling environment that complies with the XML Schema used for the XML model document.[1] You can then either run the model or edit it. Running the model first requires providing appropriate data, so we will consider this first.

Adapting the model

It is possible that the model we wish to use needs no modification: we are happy with its design, it has the required degree of spatial disaggregation, etc. In this case, we simply perform the operations outlined above. However, it is much more likely that we will want to make some changes to it:

- Change some of the equations or other aspects of its internal structure
- Link it to other models at the same level (e.g. link a forest model to a hydrology model);
- Replicate the model over multiple spatial patches: e.g. get a single-point forest stand model to operate on a European-wide basis, in each grid square that contains forest.

I will consider each of these in turn. As we go through these, note that all of these operations can be performed in a diagramming mode: there is no programming involved.

<u>Changing the model</u>
The equations in the model can be changed simply by selecting the variable of interest and entering a new equation. Other changes, such as drawing new influence links between variables, are equally straightforward.

[1] Ideally, there will be a single model-description language, with a standard ontology and XML Schema. If multiple modelling standards emerge, then techniques for sharing ontologies, currently at the research level, could be used.

Most models of reasonable complexity will contain submodels. As just one example, the LPJ model used in the ATEAM project for simulating forest dynamics contains a submodel concerned with soil water dynamics using multiple soil layers. The ATEAM modeller decides to replace it with an alternative. As mentioned in the Simile proof-of-concept section of this paper, that can easily be done, by finding an alternative soil water model on the web, clearing the existing one, and replacing it with the alternative - all with nothing more than a few mouse clicks. This functionality can possibly be reproduced in other modular modelling environments. What cannot be reproduced is the flexibility in interfacing that comes from a declarative approach: if the interface between the new submodel and the rest of the model does not match, then you simply use the standard model-editing tools to make the links. In contrast, most if not all modular modelling environments are based on rigid interfacing for the modules.

<u>Linking the model to others at the same level</u>
This involves treating our model as a submodel, to be joined with one or more other submodels. Again, this is straightforward: two or more empty submodels are created, and one model is loaded into each one. Any linkages between them (presumably there are) can either be made manually, by specifying that an input of one is a function of an output of the other. Alternatively, it can be effected instantaneously for all required links, by calling up a file specifying the linkages between the two submodels.

<u>Replicating the model over spatial patches</u>
In the ATEAM context, the model we loaded could well be a single-point model - e.g. a forest stand model. We want the same model to be run in every grid square which actually contains forest, retaining the same mathematical structure but allowing each instance to experience grid-square specific values for e.g. soil parameters and climatic variables.

Simile has demonstrated the ease with which a non-spatial model can be made spatial, given a suitable design of model-representation language. It also shows that aggregating disaggregated quantities (e.g. totalling across many spatial units) is straightforward.

Providing data

Running the model requires compatible data sets. In the context of ATEAM, the data sets are the scenario files which are generated from GCMs (global change models), and other models and approaches. Assuming that the model has inputs corresponding to those in the generated scenario files, then linking the data to the model variables is simply a matter of choosing the appropriate column in the data file (as it is in Simile at the moment). Appropriate formatting of data sets will probably remain an issue, but lessened by the standardised way of representing model variables that comes with a declarative modelling approach.

In the more general case, finding appropriate data for a model would probably involve searching the web for data, in the same way that we searched the web for the model. Now, the computer knows what the inputs are for the model: these will be variables that are not influenced by other variables and do not have values assigned to them. In fact, it knows quite a lot about them, since they will be probably be marked up with information defining their dimensions (e.g. length/time) and - possibly - units (e.g. metres/sec). Additionally or alternatively, a variable might be associated with a definition in a standard controlled vocabulary (such as the CABI Thesaurus, or the FAO AgroVoc Thesaurus [see URLs]). This information, plus other information derived from the model (time base, the nature of the submodels that a variable is nested inside) could then enable the user to be automatically presented with a list of datasets available on the web which contain appropriate data. This will be possible since these datasets will include metadata headers conforming to the same ontology. The data could include species-specific parameters, time-series data (e.g. climatic records), or spatial information (e.g. rasterised information on land use or elevation for the whole of Europe). Thus, matching up the data <u>required</u> by the model and the data <u>available</u> in published data sets is potentially a straightforward operation, provided that the appropriate metadata is associated with model variables and the data sets.

Note the difference between a declarative modelling approach and a component-based approach. In the latter, the metadata associated with model variables is quite separate from the model itself. In contrast, in a declarative modelling approach the information about a variable is associated with the actual variable in the model. Indeed, some of the 'metadata' for a variable is actually inferable from information provided as part of the process of building the model - you don't need to provide it separately.

Running the model

As discussed previously, running a declaratively-represented model involves either the use of an interpreter provided within the modelling environment, or the generation of a computer program which is then executed.

In a conventional modelling environment (or 'framework'), the model exists as a program: the model is the program. All you can do is to run this program. In the declarative modelling world, the model is separate from the program generated to simulate its behaviour. A separate simulation or model-analysis program can be generated for each possible way of processing the model. It is therefore possible for one model to be:

- run on a standard desktop PC;
- passed across to some web service to run the simulation;
- generate code for a parallel computer and run the model on that;
- run the simulation backwards (to see if we can backwards predict 1970 starting off in 2000), for those models for which this is mathematically feasible;
- perform symbolic mathematics on the mathematical representation of the model (e.g. to derive an analytical steady-state solution for a simple differential-equation model).

This is a significant benefit, since in a conventional modelling approach a separate program would be needed for each task.

In ATEAM, the standard requirement is to simulate the behaviour of the model forward through time. This is done for a variety of purposes: model benchmarking, validation, and the production of results (behaviour of key indicators), which can then enter into the analysis of vulnerability. This is a computationally-demanding activity, because of the size of the model, and the need to run the models on a 10' grid for the whole of Europe and for a variety of scenarios. As mentioned in the Introduction to this paper (Section 1), declarative modelling primarily addresses the issue of managing the model itself, rather than performing simulations with the model. The runnable version of the model is a program generated from its declarative representation, and is basically the same as the (for example) a model represented as a component in a component-based modelling framework. Therefore, the benefits of having a simulation environment, supporting benchmarking, validation and other such activities, applies equally to models developed within either approach.

Stakeholder involvement in the modelling process

So far, we have considered what it would be like to undertake ATEAM-style modelling in the researcher's office. However, ATEAM is primarily concerned with the wider policy implications of the threats of global change to ecosystem services. This involves the stakeholders both in formulating the questions - what indicators are most relevant? - and as the audience for the results. Currently in ATEAM, these interactions with stakeholders do not involve computer modelling: the stakeholders are interviewed and fill in questionnaires, with the results stored in a database of sectors and appropriate indicators and presented as a paper report.

There is now a growing feeling that stakeholders need to be more actively (or interactively) involved in the modelling process itself (van Daalen *et al*, 1998; d'Aquinoe *et al*, 2002; Standa-Gunda *et al*, 2002). Why? The main reasons are that the people affected by the results of a policy-modelling process will be more likely to accept the results if they've been engaged in the process; and stakeholders frequently can make valuable contributions to the modelling effort itself. The other question is: How? Isn't modelling a technically-demanding task, requiring specialist skills that we can't expect ordinary people to possess? Building models from scratch is; but, given an appropriate interface (usually a diagram), stakeholders with no prior modelling experience can quickly learn to 'read' the diagram and to comment on the objects and relationships it contains. Maybe not at the level of the mathematical equations, but certainly at the conceptual level.

So, in the context of ATEAM:

<u>In the early stages of the project</u>, stakeholders are shown the proposed model (at some appropriate level of detail, through some appropriate filter), or possibly the proposed alternative models. They are asked to comment on the nature of the models: What's been left out? Do the models produce the outputs (indicators that are most relevant? If not, what changes need to be made to incorporate these indicators? At a qualitative level, are the modelled relationships realistic? A declarative modelling approach, with an appropriate visual interface, can make it much easier to involve stakeholders in these important questions.

<u>In the middle stages of the project</u>, the model is run in the presence of stakeholders. They can comment on the observed behaviour, and engage in a review of the model structure to explore reasons for odd results. The researcher drives the computer, but the stakeholder is making the suggestions.

<u>At the end of the project</u>, the stakeholder gets the model, ready to use with an interface that enables alternative scenarios to be readily explored. This is not possible in the current ATEAM, since each model is implemented using a different technology, and it would take a lot of work to enable others to install the model, run it, and extract the results in a meaningful fashion. With a clean separation between the model and the tools available for using it, a future ATEAM project could enable people with little computing experience to select a model through their web browser, be presented with a GUI (graphical user interface) generated on the fly for that model (including input sliders and output graphs, tables and maps), and run the model. If they wished to drill down into the model diagram and equations, they could do so.

9.3 Re-casting existing models in a declarative model-representation language

The thought experiment presented here is based on the assumption that we are living in a declarative-modelling world: all models are already represented in a declarative model-representation language, published on the web, and so on. But we don't, and they're not! So it is appropriate to ask: How do we get there from here?

We need to be clear that declarative modelling is not simply some sort of wrapper we can put around existing models. It is an entirely new way of representing the model. Therefore, existing models will need to be re-cast in a declarative language. There are several points to make:

1. Already, considerable effort has been spent in re-writing existing (programmed) models in a new language. For example, LPJ itself was originally written in Fortran, and has been totally re-engineered in C++. Clearly, some groups do consider that it is worthwhile doing this - and that is just from one procedural implementation to another. The benefits of re-casting a model in a declarative language should provide a greater incentive to undertake the task, even if only to save on the cost of maintaining and updating the existing programming.
2. If a model has been well programmed, with well-laid out equations, and suitable partitioning off of sections concerned with updating state variables, then the effort needed to re-cast it in a declarative style is not great (weeks, rather than months). Many of the equations can be lifted as they stand.
3. If, on the other hand, the existing model code is poorly coded and difficult to disentangle, then the effort required is greater, but the need is also greater. The scientific value of a poorly-programmed model is very limited, since the essence of a model as part of the scientific

process must be clarity and rigour, and this is difficult, if not impossible to ensure if it is several thousand lines of poor programming.

4. For a well-programmed model, there is a possibility of automating the conversion process. However, this will almost certainly be incomplete, and need hand-crafting to finish.
5. It is worth remembering that programmed models (or submodels) can be wrapped up as a black-box submodel in a declarative-modelling approach. This offers a transition strategy: legacy models can be kept on in this form, to be gradually replaced by declarative re-implementations over time.

If concern over the (assumed, but probably exaggerated) cost of translating existing models is a significant issue - if it is seen as a sufficient reason to not embark on a declarative-modelling approach - then we, as a community, really do need to consider whether we are prepared to continue indefinitely with the present way of working, and to forgo the benefits of a superior approach for short-term expediency.

9.4 Conclusions

A declarative modelling approach is not a magic bullet, which at a stroke removes the need for person-years of effort in the ATEAM project. Considerable work is still required to condition the scenario data sets to make them suitable for input into models, and to adapt models to fit into ATEAM's 10' grid for Europe and 1-month/1-year time base. Apart from the possibility of generating runnable versions of the models to run on high-performance computers, running simulations would still be a computationally-demanding process, limiting the number of scenarios that could be explored.

However, it would bring considerable benefits to a project such as ATEAM. Models could be found, downloaded, adapted, linked to data and prepared for running simulations in the fraction of the time. Generic tools, shared between the various ATEAM groups, would be available for editing models, exploring their structure, benchmarking, validation and production runs. Most importantly, non-researchers (stakeholders and policy-advisers) would be brought into the modelling process in a way that is not possible with programming-implemented models.

10. Realising the vision

The widespread adoption of a declarative modelling approach would involve major changes in the working practices of a large number of researchers, the reward systems for academic achievement currently in operation, and in funding priorities. Some of these we can perhaps anticipate but, as with any technological revolution, other totally unexpected consequences will emerge. In this section I offer some preliminary thoughts on some of the issues, reflecting informal discussions with colleagues. This should serve merely to open up discussion - and even then we need to accept that technological change brings about institutional change, whatever our views on the desirability of such changes!

10.1 Voluntary or mandatory adoption?

Conceivably, funding bodies could insist that modelling within the projects they fund be done according to certain standards (declarative or otherwise). Commitment to this could be a required part of the proposal, and a criterion for evaluating the project on completion. There are precedences. In the UK, funding under the current (2002) e-science initiative requires a commitment to the Globus toolkit for Grid projects, rather than (for example) the European-developed Unicore system [see URLs].

Alternatively, researchers might adopt a modelling framework just because they want to, in the same way that they will publish their papers on the web without anybody telling them to do this. For this to work, the new approach must be manifestly better than the old.

A mandatory approach, though unpalatable to some, can be justified in societal terms: taxpayers' money is being spent, and they can rightfully expect us to maximise the return on this investment, and to use this money efficiently. We would not, I think, claim that a researcher has a right to spend time programming standard statistical tests when perfectly good packages exist for doing this, so there is no reason why someone should spend months programming a model if it can be built more efficiently through some other method.

However, in practice I believe that researchers will adopt improved methods for modelling out of choice - because there are many pressures on their time, and because they wish to have maximum impact on the rest of their community. ng in the opposite direction...

10.2 The research award system

Academics, and the institutions they work for, are assessed on the basis of two main criteria: publications and research income. Currently, any new model can be written up as a paper, or even a modification of an existing model - regardless of the fact that the paper rarely gives a complete specification of the model, and hence is not reproducible without obtaining the source code.

Declarative modelling offers the prospect of making it much easier to construct models, to incorporate other people's models in one's own, and to disseminate information about the structure of models. This is likely to reduce the emphasis on the journal paper as the primary mechanism for presenting models to the rest of the community, leading to a need to find other ways of judging the contribution made by researchers who are involved in the modelling process.

10.3 The possible emergence of de facto standards

The widespread adoption of a modelling framework requires that people accept and work to a standard. There are two ways in which this standard can arise. One, considered next, is by the adoption of a *de facto* standard, resulting from the widespread use of a particular software product. The other, considered in the following subsection, is through a collaborative standards-setting process.

AutoCAD is proprietary software package for Computer-Aided Design (CAD). The AutoCAD file format was originally developed for AutoCAD itself, but the specification was released so that other CAD systems could inter-operate with it. In fact, it is now being formally promoted as an open standard by the OpenDWG Alliance [see URLs]:

> "The OpenDWG™ Alliance is an association of CAD customers and vendors committed to promoting Autodesk's AutoCAD DWG drawing file format as an open, industry-standard format for the exchange of CAD drawings. As a result, CAD data will finally be fully-accessible; CAD software will work better with existing DWG files; and developers will be able to concentrate on building products instead of reverse-engineering solutions."

If a particular declarative modelling environment (Modelica, Simile, IMA..) were to become widely used over the next few years, then one could imagine the same words being written about its file format, for the same, laudable reasons. This does not, of course, mean that the standard is 'best' in some sense: only that it has reached some sort of critical mass and it is more convenient for people to use it rather than not.

10.4 Committee-based standards development

The advent of the World Wide Web has given a huge impetus to the process of developing software standards, especially those relating to the representation of information. The W3C (the World Wide Web Consortium) [see URLs] has some 20+ committees, including representatives from the major software companies, to develop common protocols relating to web technologies - many of them relating to XML. The W3C has developed a sophisticated methodology for the standards-setting process. For example, specification documents go through a formal series of stages: Working draft, Last call working draft, Candidate recommendation, Proposed recommendation, and finally Recommendation.

In turn, the advent of XML has resulted in a large number of groups developing a markup language for specific disciplines - there are now hundreds of such languages (see http://www.coverpages.org/xmlApplications.html). Some are the initiated and promoted by a small group; many emerge from an international collaborative process. A good example is MathML (the mathematics markup language) [see URLs]. Released as a formal W3C Recommendation in February 2001, there are now some 30 MathML-compliant software products, including Mathematica and Mathcad.

The advantages of having a standard that emerges from an international committee of leading practitioners in the field are obvious: fuller awareness of the issues, and a greater likelihood that the standard is able to cope with a wide range of requirements. This is especially important in a field as diverse as ecosystem modelling, where individual groups tend to work within a particular modelling paradigm. There can also be disadvantages: the standard can end up being too sophisticated, resulting in something that takes a long time to emerge, and which is too complex for software developers to comply with.

10.5 Research infrastructure funding

National, regional and international research programmes have a poor record of promoting and funding the development of common standards and methodologies for information processing in ecosystem research. The main exception to this has been in the area of data integration and metadata - witness, for example, the sophisticated metadata standards developed for the LTER (Long-Term Ecological Research) programme [See URLs]. This lack of investment in the information infrastructure of ecosystem research is rather surprising. After all, the processing of information is central to almost all research projects, and a small investment in improving this would certainly produce an excellent return on that investment.

10.6 Intellectual Property Rights

Changing working practices - especially creating an expectation that models will be published on the web in some text-based declarative notation - will undoubtedly cause concern amongst researchers. They will feel that their intellectual property has now become public property, and perhaps subject to too-close scrutiny by others.

These are no doubt significant concerns here, some of which can be anticipated, some which can't. What we can say is that, to date, the difficulty of getting hold of and working with someone else's model seriously undermines the scientific value of the model. If a journal paper does not describe a model to the extent that someone else can exactly reproduce it (which is true of most); and if it is difficult to obtain, install, and fully follow the logic of the model; then it does not satisfy the scientific requirement of independent reproducibility. There is thus a strong claim in the name of science for enabling models to be published in their original form, in a way which enables public scrutiny.

There are, nonetheless, ways of preserving Intellectual Property Rights. First, normal copyright laws apply, just as they would to (say) a pdf file published on the web. Second, text files can have an electronic signature, generated according to a key held by some trusted body, which can ensure ownership and the integrity of a model published as, for example, an XML document. See, for example, the XML-related solutions offered by the InfoMosaic Corporation at http://www.infomosaic.net/SignVerify.asp.

10.7 Conclusions

The adoption of technological change has two faces. On the one hand, there are processes of convergence on a standard, with little or no control from above. For example, the adoption of document formats such as Word and pdf, or the Excel spreadsheet format, just happen, without any standards committee saying that this is what we should be using. The same could happen in the field of modelling frameworks, with convergence on a standard (component-based or declarative) just by the widespread adoption of one particular solution.

On the other hand, what is not so widely appreciated is that there is a great deal of standards activity going on at a lower level - for example, the design of communication protocols such as TCP/IP and HTTP, or of languages such as HTML and XML. Moreover, these initiatives bring together the major companies in the IT field (Microsoft, IBM, Sun etc), thus resulting in comprehensive standards attracting widespread support and commitment to their use. This process of standards formation is percolating up to higher levels, resulting in standards for business communication, the provision of web services, etc.

It is conceivable that the adoption of declarative modelling could operate purely according to the first model - an *ad hoc* standard resulting from the adoption of a particular software package. Conceivable, but not desirable. Any declarative modelling standard would be greatly improved by combining the insights of a range of practitioners in the field, and there would be much greater chance of its acceptance.

This requires considerable support for the standards-development process - for the establishment of a standards authority, and for the development of reference implementations, and - into the future - for the development of generic tools rather than one-off models. This requires core funding from the major funding bodies.

As with other areas of technological innovation, the consequences will be hard to anticipate and harder still to control. Probably the only thing we can say for sure is that there will be major shifts in working practice, concepts of ownership and sharing of models, and in the reward systems for academics.

11. Critical evaluation of the declarative modelling approach

Declarative modelling represents a considerable shift from conventional modelling practice. Also, there exists a considerable resource base of models implemented using traditional methods, and there are now numerous initiatives developing modelling frameworks that are not based on declarative modelling principles. In this Section, I address some of the arguments that have been raised against a declarative modelling approach, and suggest possible ways in which it can be integrated with existing methods.

11.1 Critique of declarative modelling

Limited expressiveness

The 'expressiveness' of a language refers to the range of concepts that it can handle. For example, a modelling language that does not allow the user to specify conditional expressions (if...then...else...) is less expressive than one that does. Similarly, one that does not allow object instances to be created and destroyed is less expressive than one that does.

It is generally accepted that a declarative language (whether for modelling, architectural design, or whatever) is less expressive than a conventional programming language. To a large extent, this reflects a deliberate design decision by the developers of the language and associated software. If you are designing a continuous-systems simulation language, then you would limit yourself to this domain.

The problem we face in ecosystem modelling is that there are multiple modelling paradigms in use, suggesting that we need several separate modelling languages. However, one model might require several different paradigms to be combined (e.g. spatial, size-class and individual-based modelling), and it might seem impossible to design one language that can cater for such diversity.

Simile demonstrates that, contrary to intuition, it is possible to have a single language that is capable of handling, and indeed integrating, a wide range of modelling paradigms: differential-equation, spatial, individual-based, etc. Many ecosystem models can probably be faithfully re-cast in this language.

However, it is also true that there are some modelling paradigms that were not taken into account in designing Simile's model-representation language: for example, discrete-event model, agent-based models, economic optimisation models, Petri nets, and bond graph models. Does this prove the futility of adopting a declarative modelling approach? No: for 4 reasons:

First, it is possible that, on analysis, models expressed in other paradigms can be re-expressed in terms of Simile's model-representation language.

Second, if this is not possible, it is possible that the model-representation language can be extended to enable additional modelling paradigms to be handled, perhaps through the introduction of new symbols.

Third, if the language cannot be extended, an alternative model-representation language, based on a different ontology, could be developed.

Finally, there are many examples of useful languages which are certainly not comprehensive: for example, spreadsheets, database languages, and architectural design languages. A standard model-representation language which is not comprehensive, but which is capable of handling most common modelling requirements, could still be considered to be of great utility.

Reduced computational efficiency

There is a view that, as a general rule, a declarative approach takes more computer processing time than an equivalent procedural solution. Ecosystem models can easily become computationally demanding if, for example, they have fine spatial resolution on a grid, they include large numbers of interacting objects, or they need to be simulated many times for parameter estimation or sensitivity analysis. It is therefore appropriate to ask whether a declarative approach becomes impractical for such models.

If a model is implemented in a computer program by a human programmer, then there is plenty of scope for that program to be highly efficient. There are optimised subroutine libraries and algorithms that can be used for time-critical parts of a simulation, Various programming tricks, such as the use of hash tables and techniques for handling sparse arrays, can be employed. And models can be programmed to run on a distributed-computing or parallel-computer system.

In a declarative-modelling approach, however, the executable version of a model is produced automatically, using one of two methods. The software capable of handling models expressed in the language might have their own built-in interpreter: this appears to be the approach employed in, for example, the Stella modelling environment. In this case, evaluating an equation involves two levels - processing by the software package and processing by the language in which the package is implemented - and this necessarily reduces computational efficiency.

Alternatively, the software might use a program generator to generate a program which is equivalent to the one that a human programmer would have produced for the same model. In this case, it is possible, but certainly not inevitable, that the computer-generated program will be less efficient. It is possible because the program generator might be rather crude, or not programmed to detect special cases that could be handled more efficiently. However, it is also possible that the reverse is true. The human programmer might not have the time or the expertise to implement computationally-efficient algorithms - many ecosystem research models are implemented by research scientists or by research assistants with little training in computer science. In this case, the generated program could well be more efficient - in some cases, significantly so, since it can undertake optimisations that the human programmer might not think of, or be unable or unwilling to implement. For example, the Simile program generator automatically takes variables to the outermost loop wherever possible, whereas the human programmer probably would not do this because it would significantly complicate the code.

There are two additional points that are worth making. First, it is worthwhile investing in improving a program generator for a widely-used modelling language, because the investment is recovered across many models. So we would expect the program generator for a modelling language to gradually improve over time. This would be especially the case if the program-generating software is open-source (as opposed to a proprietary commercial), so that it could accumulate the refinements and tools for handling special cases of a large community of programmers.

Second, we could have several program generators. One could be for standard, single-processor PCs. Another could be for a workstation cluster running Condor (a widely-used network management system that manages the farming-out of a task across a cluster of workstations). Another could generate code for a parallel computer. Thus, the same model could be initially developed and run on a desktop PC, then run in a more powerful environment if required.

In conclusion, it is probably theoretically true that a hand-crafted program can be computationally more efficient than an automatically-generated one. However, in practice the reverse may well be true, and the this will become increasingly so as tools for handling declarative models gradually accumulate better and better solutions.

Restriction to scientific freedom

It may be considered that the development of a standard modelling language restricts scientific freedom: researchers should be free to adopt whatever method they choose for implementing their models.

There are several issues here. First, as presented in Section 3, numerous groups are now developing integrating modelling frameworks. By their very nature, these require standards on how modules (components) are interfaced, and thus also serve to constrain the methods that modellers can adopt.

Second, there are already several examples of standards-development within ecosystem research, and many examples in other disciplines. For example, there are many initiatives to develop metadata standards (such as the LTER metadata initiative [see URLs]), and these are seen as fundamental to the progress of ecosystem research in the internet age. It certainly restricts the types of data and metadata that can be captured, but is being adopted because of the huge benefits in shareability that it brings about.

Third, there will always be some models that cannot be implemented in a declarative language, regardless of how expressive it is. No-one, I think, would argue that the conceptual design of a model - a genuine scientific issue - should forcibly be changed to fit into some standard framework. However, the situation is rather different if the model <u>could</u> been expressed within a standard declarative language. In this case, it is much harder to argue that scientific freedom is being compromised.

Fourth, the existence of a standard does not mean that everyone is obliged to use it. It could still be up to each group to decide to implement a model through programming, even if the model was capable of being represented in a standard language, and even if this took much longer and meant that they couldn't use widely-available submodels and tools. People should <u>want</u> to use a standard because of the benefits it brings.

Fifth, there is an issue of value-for-money for the taxpayer. Certainly, one would hope that researchers would want to use a standard, rather than be forced to. But funding bodies have a legitimate concern with 'bang-for-buck': the return they get in return for the funding they put in. If a group insists on using time-consuming methods, which fail to make use of existing resources (models and tools), and which produce a product that is much less shareable with the rest of the community, then they surely need to justify this to the funding body. This justification should be more than just a claim to scientific freedom, or the fact that the research assistant employed happens to be familiar with a certain programming language.

The problem of legacy code

There are already many ecosystem or part-ecosystem models in use, for example, in forest science, agronomy, or hydrology. Researchers are familiar with their code, and unwilling to contemplate re-casting the models in some other notation. This is one of the main drivers behind the current development of a component-based modelling framework by several groups: such frameworks enable (in principle) existing legacy code to be wrapped up as a component and integrated with other components, perhaps even implemented in other programming languages.

This is a valid concern. However, it is worth making two points. First, I believe (having investigated the conversion of several programmed models into Simile) that the effort required to convert a programmed model into a declarative format is less than supposed. This is especially the case if the model was well programmed in the first place, particularly with clear separation of rate and state calculations, and sound implementation of numerical integration procedures. Second, legacy models can be wrapped up as a DLL, and embedded within a declaratively-represented model, as discussed below.

11.2 Integrating declarative and component-based approaches

The situation we find ourselves faced with is this:

- There are many models in the ecological and environmental domains currently in use, implemented in a conventional programming language; and there are many groups developing modelling frameworks, mainly component-based.
- A declarative modelling approach represents a radically different approach, based on treating the modelling process as a design activity and on the development of independent tools for processing models represented in a common language.

Is any form of reconciliation or harmonisation possible between the two approaches? I believe that there are three ways in which this is possible.

A declarative modelling approach can be used to build modules (components) for a component-based framework

In a conventional component-based approach, a component is a discrete programmed entity. These are normally programmed by human programmers. However, we can also generate such programs automatically, from a declarative representation of the model structure, as Simile demonstrates. We could therefore use declarative modelling environments such as Simile as a front end for making individual components for a component-based framework. Such components would live alongside and be interchangeable with components programmed by hand.

Moreover, since developers of this program-generating tool have complete control over the generated text, they can easily wrap the text up in whatever is needed to make it into a module for some component-based system - exactly corresponding to the wrapper a human programmer would have produced.

This approach would simplify the task of creating components, since they could be developed in a visual modelling environment.

A module (submodel) in a declarative modelling framework could be a programmed component

When Simile makes a runnable version of a model, by default it generates a single compiled unit (a DLL, or Dynamic Linked Library). However, Simile allows you to specify that some submodels are to be handled as separate DLLs, in which case the complete runnable version consists of several DLLs. (This is transparent to the Simile user). It is therefore easy to envisage that a Simile model might include DLLs produced externally, so long as they conform to Simile DLL interface. The model diagram would then show these as 'black boxes' - you can't see inside them, because they have been implemented by programming, not using Simile's diagrammatic language.

Taking 'Simile' as shorthand for any declaratively-based modelling environment, we can thus envisage hybrid models, with some parts represented declaratively, and some parts by reference to a programmed module. An extreme case of this would involve using the declarative modelling environment purely as a way of joining together programmed modules, in a manner proposed by Villa (2001) in IMA (the Integrated Modeling Architecture).

This approach would provide a means of making use of existing (sub)models, while operating in a visual modelling environment. For example, there are numerous rigorously-tested models of hydrological or atmospheric processes, which could be used as they are, without re-implementation. However, it is likely that the process of integrating (interfacing) such components into a declarative modelling environment will in general not be straightforward.

Increasing declarativeness of component-based methods

Whether or not there are initiatives to develop declarative modelling within ecosystem research, it is certain that component-based methods themselves will become increasingly declarative. There are, for example, initiatives to develop XML-based notations for specifying the interfaces of COM components in a modelling framework (see http://www.cs.man.ac.uk/cnc-bin/cnc_softiam.pl). Also, many component-based approaches draw on object-oriented methodologies, for which UML (the Unified Modeling Language) [see URLs] is the standard design methodology. There now exist various CAD (computer-aided design) packages that support UML, enabling at least parts of the programming to be done without writing code.

So, conceivably, those developing component-based frameworks will gravitate towards declarative methods. It would be unfortunate if this duplicated the effort of initiatives starting from a declarative-modelling standpoint. However, it should also be borne in mind that some of the criticisms of component-based approaches, presented in Section 3.3, do not depend on the programming involved, but on inherent limitations of the approach. Simply adopting a declarative style will not in itself overcome these problems.

12. Conclusions

There are several main conclusions.

First, if the ecosystem modelling community operated within a declarative modelling paradigm, projects such as ATEAM would be radically transformed. Activities that currently involve weeks or months or programming effort (such as downloading and running existing models) could - literally - be undertaken with a few mouse clicks. Modifying existing models or building new ones would be much simpler. And stakeholders could be involved in the modelling process.

Second, this is not some hand-waving, "wouldn't-it-be-nice-if" speculation. Everything outlined here is easily feasible - even routine - with today's technology. All that is required is the will with the ecosystem research community to make it happen, some core funding to produce the standards and basic tools, and some investment in translating a representative set of models into a declarative representation.

Third, the approach is very flexible. Models can be stored on your own computer, published on the web, or held in a web-accessible database. Tools can be developed independently, in any language. There is no forced commitment to rigid interfaces for modules.

Fourth, a declarative modelling approach is an example of 'sustainable technology'. In the 70's, people were suggesting a Fortran-base modelling framework; then it was object-oriented modelling in C++; now it is Microsoft's COM technology. Programming-based standards will change, with resulting obsolescence for the intellectual investment in models implemented according to the prevailing standard. Declarative modelling represents a specific commitment to the concept of transforming models: for display, or for generating a runnable version. If standards change - if a new language becomes the standard - then the existing pool of models can be readily preserved by translating them into the new language.

Finally, declarative modelling is at the very start of its growth curve. Its most significant feature is that we don't yet know what its most significant features will be! As with HTML, so with declarative modelling. New ideas will emerge about what we can do with models, and tools implementing these ideas. Representing model structure in a 3D virtual reality (VR) system, and navigating around it using immersive VR? Use Enhanced (or Augmented) Reality technology, so that one can see the relevant part of the model structure while moving around a landscape? Develop expert systems, containing ecosystem knowledge and modelling knowledge, to help researchers design models? Automatic model simplification? Automatic generation of a narrative, based partly on model structure and partly on simulation output, to explain model behaviour? The pebble has dropped into the pond, but the ripples will through up new and fascinating patterns for many years to come.

Acknowledgements and declaration of interest

This paper was written, and some of the work described herein was undertaken, while I was on a Darwin Trust of Edinburgh Research Fellowship. I cannot adequately express my appreciation to the Trust for its support: most academics can only dream of having 3 years to pursue Ideas, with no teaching, administration - or even reports to write. I'm now an 'independent academic' (that's another name for a house husband): thanks to Marion for carrying on working while I play on my computer.

The development of Simile has been mainly funded by the United Kingdom Department for International Development (DFID) for the benefit of developing countries The views expressed are not necessarily those of DFID. Project codes were R5652 and R7635 Forestry Research Programme. Some development has also been funded by the EU Environment Programme, within the ModMED project I gratefully acknowledge the support received from these sources.

In November 2001, a startup company, Simulistics Ltd, was formed under licence from the University of Edinburgh and with the approval of the DFID/FRP Programme Manager to develop and market Simile and related products and services. I am a minority share-holder in this company.

Jasper Taylor wrote the original Simile software - many thanks to him for turning an idea into reality.

I am grateful to the ATEAM project for inviting me to be an external assessor, and for giving me the opportunity to learn so much about such an ambitious and wide-ranging project. Parts of Section 9 were based on internal ATEAM publications and discussions with ATEAM members. I would also like to thank Denis Peter (Programme Officer, European Commission - DG Research: Environment and Sustainable Development Programme), whose idea this paper was, and the ATEAM and AVEC projects for supporting its preparation. I also acknowledge the detailed feedback provided by Rik Leemans on an earlier draft, and corrections on the nature of the ATEAM project from Dagmar Schröter.

I have appreciated useful discussions with Jonathan Massheder and Alastair Davies (Simulistics), members of the MODCOM group, and many others on the subject of declarative modelling. For some reason, most people seem to disagree with me, but that has no doubt helped to hone (and moderate) my ideas.

References

Abel, D.J., Kilby, P.J. and Davis, J.R. (1994)
The systems integration problem.
International Journal of Geographical Information Systems 8: 1-12

Chung, P.E., Huang, Y., Yajnik, S., Liang, D., Shih, J.C., Wang, C-Y. and Wang, Y-M. (1997)
DCOM and CORBA Side by Side, Step by Step, and Layer by Layer
http://www.cs.wustl.edu/~schmidt/submit/Paper.html

Costanza, R. Duplisea, D. and Kautsky, U. (Eds) (1998).
Modelling ecological and economic systems with Stella.
Special issue of *Ecological Modelling* 110: 1-103

Costanza, R. and Gottlieb, S. (1998).
Modeling ecological and economic systems with Stella: Part II.
Special issue of *Ecological Modelling* 112: 81-247

Costanza, R. and Voinov, A. (2001).
Modeling ecological and economic systems with Stella: Part III.
Special issue of *Ecological Modelling* 143: 1-143

D'Aquino, P., Barreteau, O., Etienne, M., Boissau, S., Aubert, S., Bousquet, F., Le Page, C. and Daré, W. (2002)
Role-playing games in an ABM participatory modeling process.
http://www.iemss.org/iemss2002/proceedings/pdf/volume%20due/192_daquino.pdf

Daalen, C. E. van, W. A. H. Thissen, and M. M. Berk. (1998).
The Delft process: Experiences with a dialogue between policy makers and global modellers.
In: Global change scenarios of the 21st century. Results from the IMAGE 2.1 model. (Editors J. Alcamo, R. Leemans, and G. J. J. Kreileman), pages 267-285 Elseviers Science, London.

Davis, J.R. and Farley, T.F.N. (1997)
CMS: policy analysis software for catchment managers.
Environmental Modelling and Software. 1(2).

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

Dieckmann, U., Law, R. and Metz, J.A.J. (Eds) (2000)
The Geometry of Ecological Interactions: Simplifying Spatial Complexity.
Cambridge University Press, 564 pp.

Edmonds, B., Moss, S. and Wallis, S. (1996)
Logic, Reasoning and A Programming Language for Simulating Economic and Business Processes with Artificially Intelligent Agents
http://bruce.edmonds.name/logreas/logreas_1.html

Esteva, M., Padget, J. and Sierra, C. (2002)
Formalizing a language for institutions and norms
http://www.iiia.csic.es/~sierra/articles/2002/atal01.pdf

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

Forrester, J.W. (1971)
World Dynamics.
Wright-Allen, Cambridge, Massachusetts, 142 pp.

Innis, G.S. (1975)
Role of total systems models in the grassland biome study.
In: *Systems Analysis and Simulation in Ecology, Vol. III* (Ed. B.C. Patten), p. 13-47. Academic Press, 592 pp.

Maxwell, T. and Costanza, R. (1997)
An open geographic modelling environment.
Simulation Journal 68: 265-267

McCown, R. L., Hammer, G. L., Hargreaves, J. N. G., Holzworth, D. P., and Freebairn, D. M. (1996).
APSIM: A novel software system for model development, model testing, and simulation in agricultural systems research.
Agricultural Systems 50: 255-271

McIntosh, B.S., Muetzelfeldt, R.I., Legg, C.J., Mazzoleni, S. and Csontos, P. (2003)
Reasoning with direction and rate of change in vegetation transition modelling.
Environmental Modelling and Software 18: 915-927

Muetzelfeldt, R.I. (2003)
Holistic metamodelling and its relevance to QR.
In: Proceeding of QR2003: 17th International Workshop on Qualitative Reasoning (Eds. P. Salles and B. Bredeweg), Brasilia, 20-22 August 2003, pages 3-14.

Muetzelfeldt, R.I. and Duckham, M. (in press)
Dynamic spatial modelling in the Simile visual modelling environment.
In: *Re-Presenting GIS* (Ed. Dave Unwin). Wiley.

Muetzelfeldt, R.I and Massheder, J. (2003)
The Simile visual modelling environment.
European Journal of Agronomy 18: 345-358

Muetzelfeldt, R.I., Robertson, D., Bundy, A. & Uschold, M. (1989)
The use of Prolog for improving the rigour and accessibility of ecological modelling.
Ecological Modelling 46: 9-34.

Muetzelfeldt, R. and R.D. Yanai. (1996).
Model transformation rules and model disaggregation.
Science of the Total Environment 183: 25-31

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.

van Noordwijk, M. and Lusiana, B. 1(999)
WaNuLCAS, a model of water, nutrient and light capture in agroforestry systems.
Agroforestry Systems 45:131-158)

Overton, W.S. (1975)
The ecosystem modeling approach in the coniferous forest biome.
In: *Systems Analysis and Simulation in Ecology III* (Ed. B.C. Patten), p. 117-138. Academic Press, 592 pp.

Patten, B.C. (ed.) (1975)
Systems Analysis and Simulation in Ecology, vol. III,
New York: Academic Press, p.117-138, 1975, 601p.

Potter, W.D., Liu, S., Deng, X. and Rauscher, H.M. (2000)
Using DCOM to support interoperability in forest ecosystem management decision support systems.
Computers and Electronics in Agriculture 27: 335-354

Quatrani, T. (2000).
Visual Modeling with Rational Rose 2000 and UML.
Addison-Wesley.

Raj, G.S. (1998)
A detailed comparison of COM, CORBA and Java/RMI.
http://my.execpc.com/~gopalan/misc/compare.html

Reynolds J.F. and Acock B., 1997.
Modularity and genericness in plant and ecosystem models.
Ecological Modelling 94: 7-16

Rizzoli, A.E. and Young, W.J. (1997)
Delivering environmental decision support systems: software tools and techniques.
Environmental Modelling and Software 12: 237-249

Robertson, D., Bundy, A., Muetzelfeldt, R., Haggith, M. and Uschold, M. (1991)
Eco-Logic: Logic-based Approaches to Ecological Modelling
MIT Press, Cambridge MA. 243 pp.

Rumbaugh, J., Jacobsen, I. and Booch, G. (1999).
The Unified Modeling Language Reference Manual.
Addison-Wesley

Standa-Gunda W, Haggith M, Mutimukuru, T Nyirenda, and Prabhu R (2002) (Abstract only)
Beyond modeling: Using participatory modeling approaches as tools for group communication and social learning.
http://www.cifor.cgiar.org/acm/symposium-abstract.htm#11

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

Thornley, J.H.M. (1998)
Grassland Dynamics: An Ecosystem Simulation Model.
CAB International

Villa, F. (2001)
Integrating modelling architecture: a declarative framework for multi-paradigm, multi-scale ecological modelling
Ecological Modelling 137: 23-42

URLs: (Web links)

ACSL
Advanced Continuous Systems Modeling Language
http://www.acslsim.com

APSIM
Agricultural Production Systems Simulator
http://www.apsru.gov.au/Products/apsim.htm

ATEAM
Advanced Terrestrial Ecosystem Assessment and Modelling
http://www.pik-potsdam.de/ateam

CAB International Thesaurus
http://www.cabi-publishing.org/DatabaseSearchTools.asp?PID=277

COM
Microsoft's Component Object Model
http://www.microsoft.com/com/news/drgui.asp

CORBA
Component Object Resource Broker Architecture
http://www.omg.org

DCOM
Microsoft's Distributed Component Object Model
http://www.microsoft.com/com/news/drgui.asp

ECLPSS
Ecological Component Library for Parallel Spatial Simulation
http://cycas.cornell.edu/ebp/projects/eclpss/eclpss_home.html

FAO AgroVoc
http://www.icpa.ro/AgroWeb/AIC/RACC/Agrovoc.htm

Globus
http://www.globus.org

Hurley Pasture Model
http://www.nbu.ac.uk/efm/pasture.htm

ICASA
International Consortium for Agricultural Systems Applications
http://www.icasanet.org/modular/intro.html

LTER
Long-Term Ecological Research
http://caplter.asu.edu/data/metadata/

MEDALUS
Mediterranean Desertification and Land Use
http://www.medalus.demon.co.uk/

MODCOM
Modelling with Components
http://www.biosys.orst.edu/modcom/

ModMED
Modelling Mediterranean Ecosystem Dynamics
http://homepages.ed.ac.uk/modmed/

Modelica
http://www.modelica.org

Modelmaker
http://www.modelkinetix.com

Modulus
A spatial modelling tool for integrated environmental decision making
http://www.riks.nl/RiksGeo/proj_mod.htm

Powersim
http://www.powersim.com

Pspice
Electronics simulation
http://ece110-firewall.uwaterloo.ca/ece241/PSpiceGuide3a.html

REM
Register of Ecological Models
http://www.gsf.de/UFIS/ufis/ufis_proj.html

SDML
Strictly Declarative Modelling Language
http://sdml.cfpm.org/

Simile
http://www.ierm.ed.ac.uk/simile

Stella
http://www.hps-inc.com

UML
Unified Modeling Language
http://www.omg.org/uml

Unicore
http://www.unicore.org

Vensim
http://www.vensim.com

WANULCAS
Water, Nutrient and Light Capture in Agroforestry Systems
http://www.worldagroforestrycentre.org/sea/AgroModels/WaNulCAS/index.htm

WSFL Web Services Flow Language
http://www.ebpml.org/wsfl.htm

XML
eXtensible Markup Language
http://www.w3.org/XML

XSL
eXtensible Stylesheet Language
http://www.w3.org/Style/XSL

XSLT
XSL Transformations
http://www.xslt.com

Glossary

See "URLs: Web links" above for URLs.

ACSL	**Advanced Continuous Systems Modeling Language** ACSL enables the modeller to write down the differential and algebraic equations defining a model in any order, and solves the equations using numerical integration methods.
COM	**Component Object Model** The Component Object Model is a way for software components to communicate with each other. It's a binary standard that allows any two components to communicate regardless of what machine they're running on, what operating systems the machines are running (as long as it supports COM), and what language the components are written in.
CORBA	**Common Object Resource Broker Architecture** CORBA is open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Two CORBA-based programs can interoperate (work together) on different computers, regardless of programming language and operating system. http://www.omg.org
DCOM	**Distributed Component Object Model** DCOM is a protocol that enables COM software components to communicate directly over a network in a reliable, secure, and efficient manner. http://www.microsoft.com/com/tech/dcom.asp
e-science	Science over the internet. Scientists can seamlessly access computer-based resources anywhere on the internet.
Globus	The main ***Grid*** technology, widely used in the USA (where it was developed) and other countries, including the UK.
Grid	The Grid is based on technologies for enabling scientific computing resources (sensors, databases, processors and networks) to be accessed remotely by scientists without the scientist needing to be aware of where the resources are located. It is the technology that underpins ***e-science***.
HTML	**HyperText Markup Language** A language (based on similar notation to **XML**) for marking-up text so that it can be displayed in a web browser.
IBP	**International Biological Programme** The IBP aimed to coordinate a series of ecological projects conducted by scientific communities around the world, on various 'biomes' (e.g. grassland, tundra). It was based on large-scale projects featuring new, multidisciplinary research—specifically, systems ecology.
Markup language	A language which specifies the symbols that are added to a text document to add extra information about the content of that document. The best known markup language is **HTML**. Most current markup languages conform to **XML** notation.

Prolog	**Programming in Logic** Prolog is the most widely used example of a logic programming language, and implements a subset of a type of logic called first-order predicate logic. A Prolog program consists of a collection of facts and rules. Prolog programs are not run, but instead are used to answer queries through a process of reasoning with the facts and rules. See Appendix A2.3 for more information.
Simile	Simile is a visual modelling environment for ecological, environmental and biological modelling. It was developed at the University of Edinburgh, and is based on declarative modelling principles.
SVG	**Scalable Vector Graphics** SVG is an **XML**-based language for specifying vector graphics that can be displayed in web browsers.
UML	**Unified** (sometimes Universal) **Modeling Language** UML aims to help software engineers to specify, visualize and document models of software systems, including their structure and design, prior to writing programs, in order to ensure the quality of the software.
Unicore	A European-developed ***Grid*** technology. See ***Globus***.
Web service	A web service is a web-based application that can communicate with other applications over the web. For example, there is a web service which provides the current temperature in any US city.
WSDL	**Web Services Definition Language** WSDL is an **XML**-based language that documents the inputs, outputs and other attributes of a **web service**.
XML	**eXtensible Markup Language** XML is a standard form of notation that can be used to develop **markup languages** to enable information to be passed from one computer to another on the World Wide Web. See Appendix A2.1 for more information.
XSLT	**eXtensible Stylesheet Language for Transformation** XSLT is a language for specifying how **XML** documents can be transformed into some other form: different XML, HTML, or some form. See Appendix A2.2 for more information.

Appendix 1. The Modelica, MODCOM and IMA modelling frameworks

A1.1 Modelica

Modelica [see URLs] began life in an ESPRIT project "Simulation in Europe, Basic Research Working Group" (SiE-WG). It involved a number of developers of object-oriented modelling software collaborating in the development for a common specification for physical systems modelling (e.g. electrical, mechanical and hydraulic systems). It supports the idea of hierarchical composition: a component can be specified in terms of basic components (e.g. resistor, capacitor), and that component can then itself be used as a building block. Mathematically, Modelica models represent sets of differential-algebraic equations, extended to allow for discrete events (such as a relay switching off or on).

It is important to note that Modelica is a specification for a representational language: it is not a software product. Instead, particular software developers will develop tools capable of handling models specified in the Modelica language. These tools can include diagramming tools to develop a model by selecting icons representing basic components and linking them together, tools for reporting on the model in various ways, and tools for actually undertaking simulations. The Modelica language itself and many of the component libraries are free; a number of software houses have developed commercial modelling environments that comply with the Modelica specification. Modelica is capable of specifying models with hundreds of thousands of equations. It is also capable of handling array data structures, enabling disaggregated models to be handled.

Modelica essentially operates at two levels. Most users will build models by selecting and linking pre-defined components, as described above, setting parameter values appropriate to that component (e.g. resistance for a resistor). This is an intuitive method of working, matching closely to the physical act of building an engineered object by selecting physical components with the appropriate attributes. At a lower level, defining the mathematical properties of the component is a specialised activity, and not one that most modellers would engage in.

It is tempting to consider Modelica as a standard, off-the-shelf solution to the needs of ecosystem modelling. Many ecosystem models also consist of DAEs, with array-style disaggregation. The concept of composition applies equally well to ecosystems, and support of re-use of pre-built components should prevent duplication of effort between various groups.

However, my understanding of Modelica is that it is inappropriate for ecosystem modelling, for several reasons. First, specifying equations for relationships between the model variables is much more part of the modelling process in ecosystem modelling. We cannot expect modellers to work with just the set of equations entered into a library, no matter how comprehensive this might be. Second, the notion of discrete, basic building blocks is much more appropriate for modelling engineered systems than for natural systems. Third, Modelica currently has no mechanism for specifying dynamically-varying populations of objects - an important requirement for ecosystem modelling.

A1.2 MODCOM

MODCOM [see URLs] is being developed jointly by workers in Oregon State University (USA) and Plant Research International at Wageningen (Netherlands) as a framework for ecosystem modelling. It is described thus by its developers:
"MODCOM is a tool to assist modellers in creating a broad range of object-oriented simulation models by either creating reusable components, or assembling pre-existing components into

integrated simulations. ... MODCOM provides a series of pre-built components implementing these interfaces that manage many aspects of simulations – high-level simulation services, numerical integration of differential equations, data management, data analysis, and visual representation of simulation components and outputs. MODCOM is language neutral, and is built on top of Microsoft's COM specification; any language that supports COM can take advantage of MODCOM's modeling services. ... The central concept of MODCOM is that a simulation model is best constructed as a collection of simulation objects (**SimObj**s) that are organized and managed by a simulation environment (**SimEnv**) that provide a range of high-level services common to most simulations. MODCOM provides a general-purpose simulation framework by which this can occur. By creating models as well-defined components, the opportunities for reusing the components in other simulations are greatly increased. Similarly, by handling general simulation services in a generic manner, the modeller is relieved of having to provide these services in each simulation; he/she simply takes advantage of the services offered by the framework, and can concentrate instead on defining the details of the component of interest."

There are several things to note about MODCOM. The first is that it is a rare example of the development of a modelling framework by groups from different institutions collaborating together on an equal footing, and with the primary purpose of developing the framework. Other initiatives are usually led by one particular group, operating within the context of a specific modelling project. The second is that the emphasis given to the component (module): the aim is to enable multiple components to be glued together to make a model, rather than to support the development of the components themselves. Third, MODCOM is based on a programming approach: in its current form, you need to be a programmer to make a module, and you need to be a programmer to make a model out of several components. There is an intention to develop visual modelling environments for both tasks, but that is in the future.

A1.3 IMA: Integrating Modelling Architecture

The IMA is a very ambitious project aimed at developing a generic framework capable of bringing together a wide variety of modelling paradigms in a single model (Villa, 2001). The paradigms include not only conventional simulation models based on differential/difference equations, but also agent-based models, cellular automata, optimisations models, statistical models, and so on. It is motivated by the recognition that a given problem - say, one involving land use planning with complex environmental and sociological aspects, requires a variety of approaches for its solution.

IMA is heavily predicated on a declarative modelling approach. The primary role for this is to define the characteristics of the interface that individual modules present to the rest of the system. In order to allow modules to talk to each other, to the model manager, and to tools for input, output and analysis, they have metadata attached to them defining their characteristics with respect to **time**, **space** and **behaviour**. The time aspect relates to the when a module is called during the simulation time cycle. Space relates to attributes such as grid/polygon/point representation, spatial granularity and spatial extent. Behaviour defines the type of modelling paradigm for that module: System Dynamics, event-based, or whatever. These characteristics are defined in XML, and are independent of the actual implementation of the module, which can either be programmed or represented declaratively.

The IMA vision is a sweeping one. It does share quite a lot in common with standard component-based approaches (modules, a simulation object hierarchy, as simulation controller, etc), though is distinguished by its total commitment to a declarative, ontologically-rich language for characterising modules in general, and XML in particular. At the time of writing, IMT (the Integrating Modelling Toolkit - the implemented prototype of IMA) is still under development, and it remains to be seen how realisable the vision is in practice.

Appendix 2. A brief introduction to relevant technologies

This section is for those people who would like to see some flesh on the bones of this paper. If you are happy dealing with the concepts of declarative modelling, and have no particular interest to see how these end up in practical implementations, then you can skip this section. If, on the other hand, you would like to know how the ideas are actually implemented, then this section should help.

I will consider the following technologies:

XML is the eXtensible Markup Language. It is a standard form of notation for representing information so that it can be passed from one computer to another on the World Wide Web.

XSLT is a the eXtensible Stylesheet Language Transformations. Its primary purpose is to enable XML documents to be transformed into other XML documents, HTML documents, or some other form of presentation.

Prolog stands for Programming in Logic. It is a declarative language based on first-order predicate logic, and can be used for both representing knowledge and for reasoning with knowledge.

Why did I choose these three technologies?

XML is rapidly emerging as the standard way of representing structured information on the web. There are now 100s of 'markup languages', in areas as diverse as chemistry, systems biology, architecture, literature and mathematics. Each represents the result of a group of people in a subject area getting together and deciding on what they want to represent about their field and how it should be represented. It is inevitable that any initiative to develop a standard notation for the declarative representation of ecosystem models will be based on XML.

XML by itself is of little use, unless there is some way of processing the information represented in an XML document. One way of doing this is to write a program in any common programming language: they all have APIs (Application Programming Interfaces) for working with XML documents. This gives you considerable flexibility, but does involve programming. **XSLT**, on the other hand, enables people to transform XML documents into, for example, attractive HTML documents, viewable in a web browser, without doing any programming in the conventional sense. In the present context, it thus opens the door for the development of a wide range of useful tools for displaying ecosystem models in meaningful ways.

The types of reasoning with model structure you can do using XSLT are limited to fairly basic transformations - for example, transforming the complete model specification into a document containing just the names and variables of each submodel, for displaying. Often, however, we wish to do much more sophisticated reasoning with the model structure. For example, we may wish to find the names of all variables influenced directly or indirectly by a certain parameter. Or we may wish to compare the structure of two versions of the same model. This involves logic-based reasoning with the model structure, something that **Prolog** is well-suited for.

A2.1 XML

Most people are familiar with HTML, the HyperText Markup Language. If not, just go to a simple web page in your web browser, and select **View > Page source** in Netscape or **View > Source** in Internet Explorer. In HTML, the content of the document is enclosed in start and end tags, which usually denote how it should appear. Thus:

```
<body>
<p>This is <b>my</b> <i>lovely</i> web page</p>
</body>
```

denotes the body of an HTML document, consisting of a single paragraph (enclosed in the <p>...</p> tags), with the word "my", enclosed in the <b>...</b> tags, shown in bold, and the word "lovely" shown in italics. This document has 4 **elements**, enclosed by the tag start/end pairs: note that the elements can be nested, one inside another. In HTML, the tags have a defined meaning in the HTML language, and largely relate to appearance of the content of the document, i.e. how it should be displayed in a web browser.

An XML document also consists of elements enclosed in start/end tags. However, there are two fundamental differences. In XML, the tags have **no predefined meaning**: it is up to the person marking up a document to choose appropriate tags. Usually, 'appropriate' means that the tags would have been agreed by some community of people that will be sharing the documents. Second, the tags in XML usually relate to **content** rather than **appearance**. To illustrate both points, consider how the following sentence

The bank vole, Clethrionomys glareolus, lives in hedgerows and eats berries, nuts and seeds.

might be marked up in an XML document:

```
<species_description>
The <english_name>bank vole</english_name>, <latin_name>Clethrionomys
glareolus</latin_name>, lives in <habitat>hedgerows</habitat> and eats
<food>berries</food>, <food>nuts</food> and <food>seeds</food>.
</species_description>
```

Here we have 4 types of element, species_description, english_name, latin_name, and food. These have no pre-defined meaning: presumably a group of people interested in transferring species descriptions got together and agreed on them. The elements do not tell us how the document should appear, but rather add meaning to the content of the document: for example, they enable some program to extract all foods.

And that's more-or-less all you need to know about XML. A few extra rules apply: you must have an XML header line; you must only one top-level element; all start-tags must have a matching end-tag. Also, the values of any attributes (inside the start tag) must be enclosed in quote marks.

We can use XML notation in 3 ways:

<u>Marking up continuous prose</u>
This has been illustrated by the example above. The markup is used to structures (such as the Results section of a journal paper) and individual items (such as species name) in normal text.

<u>Marking up of metadata</u>
A text document can have a set of keywords at the start, which can be marked up in XML to enable targetted retrieval by e.g. computer cataloguing systems.

<u>Marking up of structured data</u>
This is emerging as the most common use of XML, and is the way it is used in almost all the XML-based markup languages being developed. It is also what is being proposed in this paper for using XML to represent ecosystem models. As an example, the above species description can be presented in a more structured way, as:

```
<species_description>
   <english_name>bank vole</english_name>
   <latin_name>Clethrionomys glareolus</latin_name>
   <habitat>hedgerows</habitat>
   <food>berries</food>
   <food>nuts</food>
   <food>seeds</food>
</species_description>
```

This is rather close to the way that the information would be structured in a database. In fact, there are proposals for representing relational database information in XML, and conversely most XML structured documents can be expressed in a relational database format. However, XML provides rather more flexibility than standard relational databases.

A2.2 XSLT

XSLT - the eXtensible Stylesheet Language for Transformations - is a standard for defining transformations on XML documents. What this actually means is best illustrated by an example.

Take the species description given immediately above, and now consider this as part of a catalogue of species. The following box gives a small but complete XML document for two species:

```
<?xml version="1.0"?>
<catalogue>

    <species_description>
       <english_name>bank vole</english_name>
       <latin_name>Clethrionomys glareolus</latin_name>
       <habitat>hedgerows</habitat>
       <food>berries</food>
       <food>nuts</food>
       <food>seeds</food>
    </species_description>

    <species_description>
       <english_name>wood mouse</english_name>
       <latin_name>Apodemus sylvaticus</latin_name>
       <habitat>woodland</habitat>
       <food>berries</food>
       <food>nuts</food>
       <food>seeds</food>
       <food>insects</food>
    </species_description>

</catalogue>
```

Now let us say that we would like to display information on the habitat of each species inside your web browser using the following HTML:

```
  <HTML>
    <head><title></title></head>
    <body>
      The bank vole lives in hedgerows.<br/>
      The wood mouse lives in woodlands.</br>
    </body>
  </HTML>
```

When you view this HTML in your browser, you will see:

```
The bank vole lives in hedgerows.
The wood mouse lives in woodlands.
```

We need to generate the HTML by transforming the original XML. We do this using the following XSLT document:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns="http://www.w3.org/1999/xHTML">

          <xsl:template match="/">
              <HTML>
                  <head><title></title></head>
                  <body>
                    <xsl:apply-templates select="catalogue"/>
                  </body>
              </HTML>
          </xsl:template>

          <xsl:template match="catalogue/species_description">
              The
              <xsl:apply-templates select="english_name"/>
              lives in
              <xsl:apply-templates select="habitat"/>.
              <br/>
          </xsl:template>

</xsl:stylesheet>
```

Don't worry about the shaded parts at the top and bottom.   The main part consists of two XSLT **template** elements.   The text in bold relates to XSLT, the rest is sent to the output file just as it appears.   The first **template** element prints out some HTML to the output document, and calls the second template.   This second **template** prints out some English words plus the content of the **english_name** and **habitat** elements in the original document. It does this for each **species_description** element in the original XML document.

Note that an XSLT document is also expressed in XML syntax!

A2.3 Prolog

Prolog (**pro**gramming in **log**ic) is a computer language developed in the 70's for representing and reasoning with statements expressed in first-order predicate logic, a form of logic that is both expressive (it can handle a rich variety of logical statements) while at the same time having well-defined formal properties.   Prolog is a declarative programming language: rather than giving the computer instructions, as in a conventional procedural or imperative programming language, you state facts and rules about some area of knowledge.   Instead of 'running a program' to solve a problem, you 'enter a query', and the Prolog interpreter then tries to use the facts and rules to solve the problem.

Here is an example Prolog program:

```
eats(bank_vole,seeds).
eats(bank_vole,berries).
eats(wood_mouse,seeds).
eats(wood_mouse,berries).
eats(wood_mouse,insects).
herbivore(X):-
      eats(X,seeds).
herbivore(X):-
      eats(X,berries).
carnivore(X):-
      eats(X,insects).
```

You probably don't think of this as a 'program'. It contains facts that you would perhaps normally expect to see in a database, and rules. The first rule, for example, is read in English as "X is a herbivore if X eats seeds", with the **:-** symbol being read as the word "if". We certainly can't 'run' this program. But we can ask it questions, by entering a Prolog query in response to the ?- prompt produced by the Prolog interpreter. Thus:

```
      ?- eats(bank_vole,X).
```
In English: "What does the bank_vole eat?"

returns the answer

```
      X = seeds

      X = berries
```

while the question

```
      ?- carnivore(X).
```
In English: "What is a carnivore?"

returns the answer

```
      X = wood_mouse
```

using the last rule and the 5th fact.

INDEX