Ecological Modelling, 46 (1989) 9–34
Elsevier Science Publishers B.V., Amsterdam – Printed in The Netherlands

THE USE OF PROLOG FOR IMPROVING THE RIGOUR AND ACCESSIBILITY OF ECOLOGICAL MODELLING

R. MUETZELFELDT

Department of Forestry and Natural Resources, University of Edinburgh, Kings Buildings, Mayfield Road, Edinburgh EH9 3JU (Great Britain)

D. ROBERTSON, A. BUNDY and M. USCHOLD

Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN (Great Britain)

ABSTRACT

Muetzelfeldt, R., Robertson, D., Bundy, A. and Uschold, M., 1989. The use of Prolog for improving the rigour and accessibility of ecological modelling. *Ecol. Modelling*, 46: 9–34.

We introduce three concepts that offer considerable benefit to the process of ecological modelling: the descriptive representation of models; the explicit representation of knowledge about how to model; and the development of knowledge-based systems that can help ecologists construct models. Prolog, a computer language based on formal logic, has much to offer in realising these ideas. We introduce the concept of a 'model blueprint', a complete, formal specification of the structure of a model, and show how a blueprint can be represented as a Prolog program, basing our analysis on system dynamics models for simplicity. We consider ways in which the Prolog interpreter can be used selectively to retrieve information about the model, to check for errors in the formulation of the model, and to evaluate the model mathematically. However, there are drawbacks with this approach, so we discuss ways of overcoming these by implementing – also in Prolog – programs which buffer the user from the difficulties of working at the level of the Prolog interpreter. These include the generation of descriptions of model structure, and the development of a program to help in the construction of simulation models.

1. INTRODUCTION

This paper is about three radically new concepts in the field of ecological modelling. These are based on recent developments in artificial intelligence, particularly in the field of knowledge-based systems, and offer the possibility of greatly improving the rigour of the modelling process and the accessibility of modelling to ecologists.

0304-3800/89/$03.50 © 1989 Elsevier Science Publishers B.V.

The first concept deals with the way that models are represented. Currently, most models are implemented in some programming or simulation language, and described in a scientific paper. This has a number of problems: there is nothing automatically to guarantee identity of the two versions of the model; the textual description may be (and often is) incomplete; and the program listing rarely provides an easily-understood description of model features. To get around these problems, we have developed the concept of the 'model blueprint', a complete description of the structure of a model that can serve both to generate an implementation of the model as a runnable program, and can also serve to generate descriptions of any aspect of the whole or part model. Loehle (1987), in speculating on possible applications of artificial intelligence to ecological modelling, does not appear to view model structure as a set of information: to us, that is the core upon which other modelling-related activities are based.

The second concept deals with the way the modelling knowledge is represented. The term 'modelling knowledge' covers all aspects of the expertise that a modeller brings to bear when constructing a model: for example, the ability to detect erroneous or incomplete models; the ability to decide on what to include in a model given the context and the objectives of the modelling exercise; and the ability to choose an appropriate modelling formalism for the problem in hand. Unfortunately, any one modeller is unlikely to have full understanding of all relevant principles, given the variety of modelling approaches available and the variety of ecological systems. Therefore, we propose the concept of the 'modelling rule-book', a formal representation of modelling knowledge that would serve not only as a source of information about how to model, but can also be used to check models and to advise on their construction. Such a common modelling knowledge base would, being explicit, have the tremendous benefit of enabling the consistency of different modellers' approaches to be assessed.

The third concept deals with the way that models are constructed. Modelling is inaccessible to many ecologists. They may have a sound ecological understanding of some problem and the ecological system in which it is embedded, but lack the mathematical, modelling and programming skills necessary to construct appropriate simulation models. The Edinburgh ECO project has been concerned with the development of a number of model-building environments that enable ecologists to construct models using ecological terminology, rather than being forced to think in computing terms (Uschold et al., 1985; Muetzelfeldt et al., 1986, 1988). Such systems involve the combination of a user interface, the modelling rule-book and an appropriate model-building strategy, and generate model blueprints as output.

This paper has two aims. The first is to demonstrate the practical feasibility of the above ideas by giving examples in the logic programming language Prolog. The examples given here are highly simplified versions of the more complex systems that have developed in the ECO project. The other aim is to provide sufficient examples so that the interested reader – with access to Prolog and a willingness to learn the rudiments of the language – can try out the ideas for him or her self. It should thus be apparent that we are not claiming to give definitive solutions in this paper, but rather to illustrate important aspects of the solutions.

We have chosen Prolog as the vehicle for expressing our ideas for a number of reasons. First, as a language based on first-order predicate logic, it is recognised as being a powerful 'knowledge representation formalism' (Malpas, 1987): other formalisms, such as frames and semantic networks, can be readily implemented in Prolog. Second, Prolog programs have a very simple structure, and are very readable once the few basic elements of the language have been assimilated. Third, Prolog supports a declarative programming style, in contrast with conventional, procedural programming languages. This means that the programmer's task is generally to state what is true rather than to tell the computer how to solve a problem, so the programmer can leave many of the control decisions to the Prolog interpreter (the inference mechanism that is able to reason with the statements in the program). Fourth, Prolog can be used to develop metalevel code, that is a program which can manipulate a Prolog program. This is a very powerful facility which makes it possible, for example, to enable a program to modify itself during the course of execution, and to extend the inference mechanism provided by the standard Prolog interpreter.

For the interested reader who wants to learn more about Prolog, and try out some of the examples given in this paper, there are a number of suitable textbooks. Burnham and Hall (1985) and Rogers (1986) are simple introductory texts. Bratko (1986), Sterling and Shapiro (1986) and Malpas (1987) deal with more advanced aspects of the language.

We begin by showing how Prolog can be used to represent the model blueprint for models that conform to the formalism of system dynamics (see Wolfe et al., 1986). We show how the Prolog interpreter can be used to answer questions about the structure of the model, to check for errors in the model, and to evaluate the model numerically. We then discuss the deficiencies of this approach, and show that Prolog provides the features of a conventional programming language for improving the user interface and computational efficiency, without losing the logical basis of the underlying model representation. We consider in particular the display of models, the development of a knowledge-based model construction program, and the simulation of model behaviour.

2. REPRESENTING SYSTEM DYNAMICS MODELS IN PROLOG

System dynamics is a modelling formalism that has been widely applied for modelling ecological systems. Wolfe et al. (1986) give a short introduction to the approach, with source references, and give an example of using a system dynamics approach to modelling the dynamics of a pond. Although it has considerable limitations as a general formalism for modelling, it forms an excellent starting point for considering the formal representation of ecological models, since it has a small number of modelling elements (compartments, flows, external variables, etc), and it has a small set of simple and easily-understood symbols which enable system dynamics models to be displayed diagrammatically.

We use the term 'model blueprint' to describe the formal representation of model structure. The blueprint is central to other modelling activities: it describes the model, so enables the model to be displayed in various forms; it can be checked for errors in model formulation; it is the structure

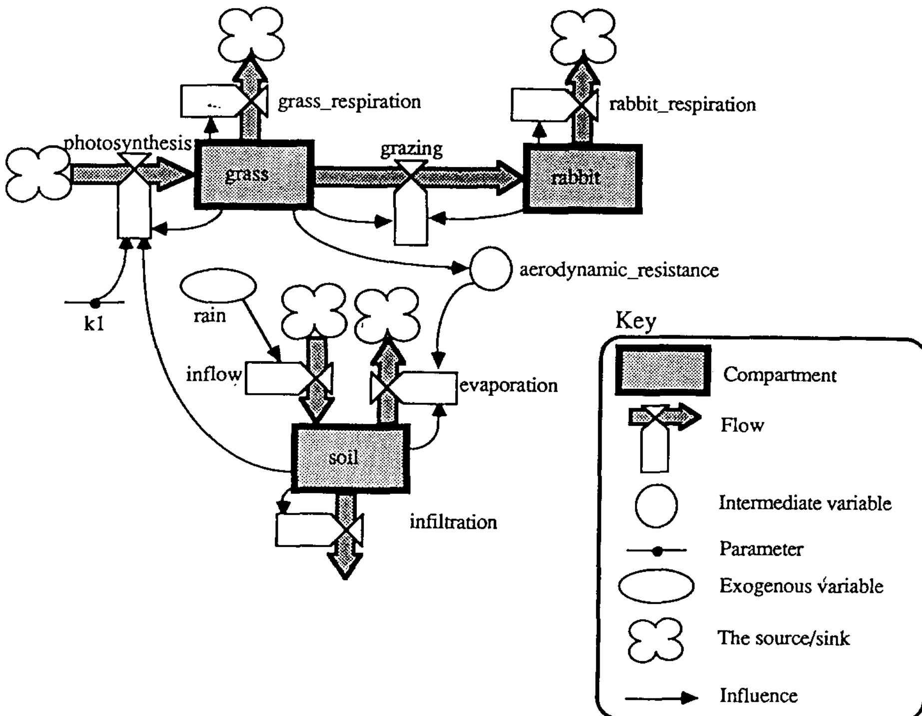

Fig. 1. Diagrammatic representation of a hypothetical, three-compartment system dynamics model. The top half represents levels and flows of biomass, the bottom half represents the level and flows of water in a single soil-water compartment. Note that only a single parameter (k1) is shown for clarity.

generated by a model-building program; and it can be used to generate an implementation of the model in a conventional programming or simulation language. The blueprint consists of a set of Prolog statements, which together specify all relevant properties of the model. Information is included in the blueprint either because it is necessary to complete the mathematical specification of the model, or because it is useful in providing subsidiary information about the model.

Figure 1 is a schematic representation, using system dynamics notation, of a simple hypothetical model of the flow of biomass in a grazing system, with the photosynthetic rate of grass influenced by soil moisture. As with any system dynamics diagram, there are two inter-linked networks: the flow network, and the network of influence relationships.

2.1. *Representation of compartments and flows*

In considering how to represent the structure of this model in Prolog, we begin by considering the flow network, which includes 'compartments', and the 'flows' between compartments. The following Prolog statements name the three compartments, and specify the compartments involved in three of the flows:

```
comp(grass).
comp(rabbit).
comp(soil).
flow(photosynthesis, outside, grass).
flow(grazing, grass, rabbit).
flow(evaporation, soil, outside).
```

This is a short Prolog program, consisting of six 'clauses'. Each of these clauses is a 'fact'. The terms *comp* and *flow* are 'predicates'; note that the predicate names have no built-in meaning (we choose names that we find descriptive). The *comp* predicate has one 'argument', while the *flow* predicate has three – the flow name, and the donor and recipient compartments. The description of the flow network is completed by adding more *flow* clauses.

One of the main advantages of the Prolog approach is that it is easy to add extra information about the model as required – information that does not relate directly to the mathematical structure of the model, but which provides useful information about it. For example, the substance associated with each compartment can be represented by statements of the form:

```
substance(grass, biomass).
substance(soil, water).
```

2.2. *Representation of influence relationships*

The other main network in the system dynamics framework is made up of the influence relationships used to calculate (for example) the rates of various processes (Fig. 1). Each link in this network represents an influence of one quantity on another. One way of handling these influence relationships is by making a separate assertion for each link:

influences(grass, aerodynamic_resistance),
influences(rabbit, grazing).
influences(k1, photosynthesis).
etc

Each of the two arguments is a parameter or one type of system dynamics variable, as described below.

2.3. *Representation of variables*

The system dynamics methodology conventionally includes five types of quantity:
- state variable (also known as level, compartment)
- rate variable (flow, process)
- intermediate variable (auxiliary variable, endogenous variable)
- exogenous variable (external variable, forcing function, driving auxiliary)
- parameter (constant, coefficient).

It is natural to include statements in the model blueprint identifying each type of quantity, e.g.

state_variable(rabbit).
rate_variable(grazing).
intermediate_variable(aerodynamic_resistance).
exogenous_variable(rain).
parameter(k1).

While there is nothing wrong with this, we can in fact infer the type of each quantity from other information in the blueprint, providing a much more concise and elegant way of representing the same information. In order to do this, the blueprint needs to contain 'rules' in addition to the facts it contains already. For example, the following rule defines a state variable:

rate_variable(X):-flow(X,_,_).

X is a 'Prolog variable', while the underscore character ('_') stands for arguments that are not of interest. The symbol ':-' is read as 'if', so we read the whole rule as 'X is a rate variable IF X is the first argument of a flow predicate'. This therefore enables the deduction to be made that e.g. grazing is a rate variable, eliminating the need for many such facts.

2.4. *Assigning values to variables and parameters*

There are two aspects to the numeric side of a simulation model. First, numbers form part of the description of a model, and therefore should be included in the completed model blueprint: parameter values, initial values for state variables, and time series data for exogenous variables. Second, values are of course calculated during the course of a simulation for state and intermediate variables.

One way of representing this numeric information – whether assigned initially or calculated during the execution of the model – is to have a single predicate, say *value*. It takes three arguments: the name of the quantity (variable or parameter); its actual value; and the time at which the value applies.

The *initial value for state variables* can be represented by statements of the form:

value(rabbit, 10, 0).

meaning that the biomass of rabbits is 10 at time 0. We will discuss later (in Section 3.3) how to specify the value of state variables at other points in time.

The *value of parameters* can be represented by statements of the form:

value(k1, 0.005, T).

meaning that the value of the parameter $k1$ is 0.005 at any (all) time (since T is a Prolog variable).

For *exogenous variables*, we need to specify the value at each point in time. Thus, for a variable which is a tabulated function of time (as, for example, a record of daily rainfall), we make a series of assertions, such as:

value(rain, 15, 0). meaning: "the rainfall on day 0 is 15"
value(rain, 17, 1).
value(rain, 18, 2).
value(rain, 17, 3).
etc

Alternatively, it will sometimes be possible to use a few rules instead of many facts:

```
value(rain, 0, T):-T < 20.
value(rain, 5, T):-T > = 20.
```

which causes the exogenous variable *rain* to take the value 0 before time 20 and 5 thereafter. Moreover, since Prolog allows facts and rules to be mixed, there is considerable flexibility in the representation of time-dependent information.

There are a number of possible strategies for representing functional relationships (i.e. the formulae for calculating the values of 'intermediate variables' and 'rate variables'). The methods differ in how readable they are as Prolog code, and the extent to which they support the two main functions of the model blueprint – providing a description of the model, and permitting efficient numerical simulation of the model. One possibility is to maintain two versions of each formula: one as text suitable for describing the relationship, the other in a procedural form suitable for numerical evaluation. However, this is undesirable, since it means that potentially the two forms may not correspond.

We can define a predicate *formula* which has three arguments: the quantity calculated from the expression, the expression itself, and time (for consistency with the *value* predicate). For example:

```
formula(photosynthesis, k1 * grass * soil, T).
formula(grass_respiration, 0.02 * grass, T).
```

This representation is very straightforward, and is easy to understand when obtaining descriptions of the model. However, in order to support numerical evaluation of the model, it is necessary to write a *value* rule for finding the values associated with the named quantities, and evaluating the resulting numerical expression:

```
value(Variable, Value, T):-
    formula(Variable, Formula, T),
    expand(Formula, Expression, T),
    Value is Expression.
```

in which the predicate *expand* produces an expression containing the corresponding numeric values. Values obtained by evaluating formulae are then obtained in a manner completely consistent with that used for all other numeric quantities.

3. USING THE PROLOG INTERPRETER TO ANSWER QUESTIONS ABOUT THE MODEL

Once the model blueprint has been captured as a set of Prolog clauses, we are in a position to use it in various ways: for example, to answer questions about the structure of the model, to check for errors in the formulation of the model, and to generate numerical solutions of the model. In this section, we demonstrate the use of the standard Prolog interpreter to do these jobs. In the next section, we discuss how the user interface can be improved to overcome some of the deficiencies of the 'raw' Prolog approach.

3.1. *Asking questions about the model structure*

One of the justifications for holding blueprints as a set of data is to enable its structure to be interrogated and displayed. This enables a focussed investigation to be made through a complex model, and comparisons to be made between a variety of models held in a common model library, assuming that all the blueprint clauses are tagged with an extra argument specifying a particular model.

First, we can ask questions that are answered directly from the information provided. To do this, we enter our queries into the Prolog interpreter, which then reasons with the facts and rules making up the model blueprint, and displays a response. For example, to find out what compartments are in the model, the user could type:

?–comp(C).

The computer replies with:

```
C = grass;
C = rabbit;
C = soil;
No
```

with the user typing a semi-colon after each answer to force the search for other answers, until no more can be found.

Similarly, questions can be asked about any other aspect of the model:

?–influences(V, grazing).	asks what variables (*V*) influence grazing.
?–flow(F, grass, _), *formula(F, Form, _).*	ask about the formula (*Form*) associated with each flow (*F*) out of the grass compartment.
?–flow(F1, grass, C1), *flow(F2, C1, C2).*	asks for the compartments (*C2*) that are connected by two flow links, via *C1*, to grass.

As it happens, these questions can be answered directly from facts in the model blueprint. Other questions might involve more complex reasoning – for example, with rules that enable the direct or indirect influence of any quantity on any other quantity to be inferred.

3.2. *Detecting errors in the model blueprint*

Given a formal representation of a model, it is reasonable to want to detect errors in the blueprint. The errors can be of various types:
- At the simplest level, we want to be able to detect errors of 'modelling syntax'; for example, that we do not have a direct influence of a variable on a compartment.
- At a higher level, we may wish to ensure that the model is consistent with modelling principles; for example, that the degree of disaggregation or time constants do not vary greatly in different parts of the model.
- Beyond that, we may wish to make sure that the model makes 'ecological sense'; it would be a mistake, for example, to have a flow of biomass from rabbits to grass, or a circular flow loop for biomass.
- Finally, and most demanding of all, is to ascertain that the model is appropriate to the objectives of the modeller.

Prolog rules offer a simple mechanism for detecting errors in a model blueprint. Furthermore, as we shall see in a later section, the same rules can be used to guide the construction of a model, and thus provide the basis for an intelligent model-building assistant.

In this section, we will limit ourselves to checking the modelling syntax and the ecological sense of system dynamics models. For both types of check, it would be desirable to distinguish between an actual error message and a mere warning, but this distinction makes no difference to the Prolog approach.

Syntax: does the model accord with the conventions of system dynamics? A simple way of providing the user with an error-checking mechanism is to enable him or her to enter the query:

```
?-error(A, B).
```

in which *A* is the type of error and *B* is the model element (or possibly a list of model elements) involved in the error.

The predicate *error* is then defined by a number of rules, one for each type of error. For example, it would be an error to have a flow between two compartments which do not contain the same substance. This can be

detected with the rule:

```
error('Incompatible flow', F):-
    comp(C1), comp(C2), flow(F, C1, C2),
    substance(C1, S1), substance(C2, S2), not(S1 = S2).
```

Does the model make ecological sense? Detecting errors in the ecological sense of a model is a much more open-ended task. First, it is necessary to provide an ecological knowledge base, containing information ranging from the characteristics of species in particular areas through to the definition of ecological concepts. Second, it is necessary to ensure that the names of ecological elements in the model (e.g. compartments and flows) correspond precisely to the names used in the ecological knowledge base, since otherwise it is not possible to check the model structure against the store of ecological information.

The representation of ecological knowledge is a wide-ranging topic which can only be touched on here: it involves the representation of taxonomic relationships, ecological relationships between species, knowledge on the spatial and habitat distribution of species, etc. There are difficult issues involved in deciding how to represent different types of knowledge, and this is an important area of research in artificial intelligence. Furthermore, it must be accepted that the task is an open-ended one: there will always be some new ecological study that can be undertaken which is original in space, time, or the species involved. However, what is at issue here are the basic principles involved in representing ecological knowledge for checking models.

Prolog facts and rules offer one possible method for representing certain types of ecological knowledge, a method which is compatible with the representation of model structure and rules of syntax. In the examples that follow, we will restrict ourselves to rules represented directly in Prolog, and handled with the basic Prolog interpreter. However, we reiterate the point emphasized in the Introduction, that this approach is chosen for its simplicity as an educational device, and as a starting point for those with access to a computer running Prolog. Simple rules plus the Prolog interpreter do *not* provide a comprehensive solution to the problems of representing and reasoning with ecological knowledge, and there remain difficult issues in both these areas.

In the examples that follow, we limit ourselves to making statements about particular groups of organisms, corresponding to the compartments that might be used in developing a System Dynamics model of some ecological system. There is a certain looseness in this approach, since the ecological statements about *rabbit* refer to a particular taxonomic group,

whereas the compartment *rabbit* refers to the population in a certain area. This distinction can be glossed over here, but would need to be accommodated in a fuller development of a knowledge-based modelling system.

As an example of ecological knowledge that can be useful in checking models, we can consider feeding relationships. These are generally pictured as a food web, and are readily represented in Prolog. We can define the predicate:

eats(A, B).

to mean that the taxonomic group *A* feeds on the taxonomic group *B*. The following two assertions then represent part of a much larger food web:

eats(fox, rabbit).
eats(rabbit, grass).

These statements can be built on to define key ecological roles, such as *herbivore, carnivore*, etc.:

herbivore(A):–	"Group *A* is of type herbivore IF
eats(A, B), plant(B),	it eats something (*B*) which is a plant, AND
not(eats((A, C), animal(C))).	it does *not* eat something else (*C*) which is an animal"

We are now in a position to detect ecological misconceptions in the model structure, provided that the names of compartments and flows in the model correspond to those used in the ecological knowledge base. For example, assume that the model blueprint contains the statement:

flow(grazing, grass, fox).

The following rule would report this as an error, since there is a flow of biomass between grass and something which is not a herbivore:

error('Unrealistic flow', grazing):–
flow(grazing, C1, C2), not((plant(C1), herbivore(C2))).

This states that it is an error to have a grazing flow between *C1* and *C2*, if it is not true that *C1* is a plant and *C2* is a herbivore.

This rule holds an important lesson for the development of practical knowledge-based modelling systems, as opposed to the simple, illustrative systems presented in this part. This rule will return an error message EITHER if it is known that *C1* is not a plant, OR if it does not know about *C1* at all. In order to get around this, more subgoals would need to be added, to check

that the types of *C1* and of *C2* were actually known, and the error only detected if the types of *C1* and *C2* were *known* to be wrong. Now we could certainly do this, but at the expense of reducing the 'clean-ness' of the rules, since they would be cluttered up with various checks. Rather, we should seek to extend the power of the interpreter, so that checks of this type would be done at the level of the interpreter, *not* at the level of the application-specific rules. The interpreter would thus no longer be the Prolog interpreter itself, but an extension of it which was more suited to the particular requirements of ecological modelling. Fortunately, the Prolog language enables such enhanced interpreters to be written in Prolog itself.

3.3. *Evaluation of variables and running the model*

We have seen that the Prolog query mechanism can be used to find out about aspects of the model structure. Asking questions about the initial value for a state variable, the value of some parameter, or the value of an exogenous variable at any point in time is really part of the same activity, since these are all part of a completed model blueprint. However, we have suggested above that the *value* predicate can be used to capture all the numeric quantities of a model, at any point of the simulation. Therefore, it should be possible to answer questions involving the predicate *value* by numerical evaluation of the model, as well as by direct reference to the way the model was set up.

The general query:

?–*value(Variable, Value, Time).*

finds the value of some (model) variable or parameter at some point in time. Thus, we can find the values of parameters and exogenous variables at any time, through a query such as:

?–*value(kl, V, _).*	What is the value *V* of the parameter *k1*?
?–*value(rain, V, 10).*	What is the value *V* of rainfall at time 10?

Similarly, the initial value of a state variable can be found by asking a question such as:

?–*value(rabbit, V, 0).*	What is the initial biomass of the rabbits?

making sure to set the time to zero.

Furthermore, assuming that functional relationships have been implemented in a manner compatible with the *value* predicate, as suggested

above, then the values of intermediate and rate variables can be determined at time zero with the same type of question:

```
?-value(grazing, V, 0).
```

Simulating the dynamic behaviour of the model involves determining the value of one (or more) state variables (corresponding to compartments) at any point in time. In order to do this, we need rules which can determine the value of a state variable from its previous value, plus or minus the flows affecting it. The following is one possible solution, in which the state variable *Variable* takes the value *Value1* at time *Time1*, and *Value* at the preceding point in time, *Time*. This is a *general* solution, requiring only the definition of the two predicates *inflow* and *outflow*, respectively, whose job is to total up all the inflows into, and all the outflows out of, the compartment under consideration:

```
value(Variable, Value1, Time1):-
    state_variable(Variable),
    Time is Time1-1,
    value(Variable, Value, Time),
    inflow(Variable, Time, Inflow),
    outflow(Variable, Time, Outflow),
    Value1 is Value + Inflow-Outflow.
```

Given this representation, we can now formulate a query in Prolog in terms of the value of any variable at any point in time. For example:

?-value(rabbit, V, 30).	What is the value (V) for the biomass of rabbits at time 30?
?-time(T), T < 30, value(rabbit, V, T).	Find the rabbit biomass for all times from 1 to 30 (assumes that *time* has been defined to give $T=1$; $T=2$; etc).
?-time(T), value(rabbit, V, T), V > 100.	At what time (T) does rabbit biomass exceed 100?

These examples demonstrate that the uniformity of the approach to the handling of time offers a great deal of flexibility in how the model can be used. With some modification to the model representation, this approach could even allow the model to be run backwards as well as forwards through time, enabling one to ask what biomass would be needed at time 5 if the biomass at time 15 is 125 $\mathrm{g\,m^{-2}}$.

The attractiveness of this approach is that asking questions about the future state of the model (i.e. simulating model behaviour) is seen as no different than asking question about the model itself: in both cases, the

Prolog interpreter uses a process of deductive reasoning with the model blueprint to yield the answer. However, there are severe penalties with this approach. In the first example (?–*value(rabbit, V, 30)*), the Prolog interpreter recurses back to time 0, in order to find the initial values for the state variables. This does not matter in this particular case, and the same result is obtained as if the set of difference equations had been solved numerically in a conventional program. But in the second example, the computer has to recurse down to 0 for *every* time from 1 to 30. This is clearly an unacceptable overhead, especially with more complex models, and with a shorter time step for integration. Moreover, if the user wanted results for more than one state variable (which is almost always the case), the model would have to be evaluated separately for each one. Furthermore, this approach does not support any interaction with the user during the course of the simulation, as one might need in, for example, a wildlife management model.

4. IMPROVING THE USER INTERFACE AND COMPUTATIONAL EFFICIENCY

Prolog is not just a language for representing knowledge as facts and rules, and answering questions using the reasoning powers of the Prolog interpreter. If it were just that, then the deficiencies noted above would limit its usefulness in the field of ecological modelling. Rather, it can also be viewed as a procedural programming language, enabling jobs to be done such as handling printing and user input. Prolog comes supplied with a set of 'built-in predicates' which support these tasks.

4.1. *Improved display of the model blueprint*

Using the Prolog interpreter to ask questions about model structure obliges the user to know the name and number of arguments for the predicates used to represent model structure, and also possibly the name given to each model element (compartment, flow, variable) if she or he wishes to refer to them in a query. It is desirable to provide an interface which removes this burden from the user. This also opens up the possibility of producing complete and easily-understandable displays of model structure.

Thus, a sample dialogue (in which the user types only the query *display* and the number *1* in response to the prompt) might be:

?–*display.*
Do you want to:
1 – List all compartments

2 – List all flows for one compartment
Your choice: *1*
The compartments in the model are:
soil
grass
rabbit

A skeleton example of a program that can work in this way is given in Appendix 1. This approach can be readily extended to generate comprehensive descriptions of the structure of a model. The description can be:
- in English, by embedding terms specific to a particular model in canned phrases;
- in mathematical notation, by printing out the differential and subsidiary equations; or
- diagrammatically using the symbolism of system dynamics, by exploiting the graphical interface available with several versions of Prolog.

4.2. *Development of an intelligent model-building interface*

It has been assumed so far that, as with any other Prolog program, the Prolog representation for a particular model is entered by an editing process, using whatever editor is provided with one's version of Prolog. Although cumbersome, this approach corresponds exactly to the way in which one would enter a Fortran program, or the statements in a conventional simulation language. As with these other languages, this approach requires the naive user to learn about (for example) the rules of syntax of the language before he can make use of it. It is clearly desirable to provide the user with an interface for entering the model representation without having to work at the level of Prolog syntax, and without having to remember the names and arguments of the predicates used to capture model structure.

Various interfaces have been developed in other modelling domains to facilitate model construction: for example, in chemical process engineering (Fjellheim, 1986) and in power system design (Fujiwara and Sakaguchi, 1986). In the area of system dynamics modelling, the Stella modelling package has been developed for the Apple Macintosh (Lewis, 1986), while Leaning and Nicolosi (1986) describe a program that helps users build compartment models in the area of human physiology.

A number of interfaces for building ecological system dynamics models have been developed within the Edinburgh ECO project (Uschold et al., 1985; Muetzelfeldt et al., 1986, 1988). One of these enables a user to enter statements which convey system dynamics information: for example, typing "rabbits graze grass" results in the setting up of two compartments and a

flow between them. In addition, we have experimented with menu-based and question-and-answer interfaces, with and without an expert system component driving the dialogue. We have also developed a 'database browser' (Robertson et al., 1985), which makes it very easy for an ecologist to search for equations, parameter values, etc, during the process of model construction. We are currently developing a program which enables models to be constructed by the selection of phrases that describe the real-world system (Robertson et al., 1988).

In this section, we show at a very simple level how Prolog can be used to develop interfaces which can help ecologists construct ecological models. We begin by considering how information supplied by the user can be stored away as Prolog statements making up the model blueprint. We then consider some requirements of a system that can guide the user through the model-building process, and finish with an example (albeit a minimal one) of a rule-based model-building program.

It is quite easy to develop an interface which:

- gives the user a choice of possible actions – "add a compartment", "add a flow", "initialise a compartment value", etc. – similar to the approach used for displaying parts of the model blueprint;
- picks up additional information, such as a compartment name; then
- adds the necessary fact or facts to the Prolog data base.

For example, the clause *build(1)* adds a new compartment to the model specification:

```
build(1):-
    write('Compartment name:'),
    read(C),
    not(comp(C)),
    assert(comp(C)).
```

On entering the Prolog query:

```
?-build(1).
```

the computer writes out the text "Compartment name:" as a prompt, the user types in a name, a check is made to ensure that it is not already in the model, and the built-in Prolog predicate *assert* adds a *comp(C)* clause to the Prolog database (and thus to the model blueprint), with *C* instantiated to the name supplied by the user. Thus, if the user had replied *sheep* in response to the *read(C)* subgoal, then *comp(sheep)* would be added.

In order to provide a menu-based interface for model-building, it is simply necessary to define a new predicate (say *build*), which prints out a

menu and gets the user's choice. This approach is identical to that used for displaying model structure, as described above.

It is much more interesting, however, to consider the active guidance of the user during the process of model construction. This topic, which is a key element of the Edinburgh ECO project, is very open-ended. Ultimately, it should result in a software package which can help ecologists who are completely naive about programming and modelling to build complex and realistic models addressing ecological, environmental and resource management problems. In this paper, however, we will restrict ourselves to considering the issues involved in setting up a simple, question-and-answer interface that helps in the building of ecological system dynamics models.

First, we can consider the various types of action that can be involved in building a model:

(1) There are things that *must* be done, and can be done automatically, e.g. adding a *comp(sheep)* clause given a flow clause referring to sheep;
(2) There are things that *must* be done, but require user input, e.g. initial value for a compartment;
(3) There are things that *must not* be done, e.g. putting an influence onto a state variable;
(4) There are things that *probably should* be done, e.g. adding a respiration flow for a compartment whose units are biomass;
(5) There are things that *probably should not* be done, e.g. a flow of a nutrient from an animal to a plant (though the plant might be insectivorous!); and
(6) There are things that *can* be done, e.g. adding compartments, flows, variables.

(1) and (2) can be handled by checking model structure after each action by the user, and handling all such outstanding jobs. (3) and (5) can be handled by putting constraints on every rule that allows something to be added to model structure, to ensure that nothing is added which would result in an illegal or dubious model structure. However, this could result in considerable complexity in a large number of rules. It would probably be better to invoke calls to *error* and *warning* before each intended action actually resulted in a change to model structure, but checking as though the change had been implemented. In other words, the model-building activity would be supervised by a separate monitor which could trap possible errors before they were incorporated into the blueprint. (4) can be handled by a suggestion mechanism, as illustrated below, while (6) can be handled by a general 'escape' mechanism which takes the user into the generalised model-building menu suggested above.

A minimal rule-based model-building program

Figure 2 gives a listing of a rule-based model-building program. It is complete, in that it should run as it stands with most Prolog interpreters. It is clearly minimal: it only contains three model-building rules, and a tiny ecological knowledge base. However, both of these could be expanded as required. This program builds a model one element at a time, with the addition of each element being obligatory or suggested. It handles action types (1) and (4) from the list above, and with simple additions can handle action types (2) and (6). Figure 3 is a transcript of a session with this program.

Part A - model-building shell

```
build_model:-
  level(Level),
  possible(Level,Element,Text),
  not(already_handled(Element)),
  handle(Level,Element,Text),
  build_model.
build_model:- listing(comp), listing(flow).

level(required).
level(suggested).

already_handled(Element):- call(Element).
already_handled(Element):- reject(Element).
already_handled(Element):- undecided(Element).

handle(required,Element,Text):-
  write('I will now '),write_text(Text),nl,assertz(Element).
handle(suggested,Element,Text):-
  write('I suggest that you '),write_text(Text),nl,
  write('OK? y. = yes, n. = no, u. = undecided'),nl,
  read(Ans),handle_answer(Ans,Element).

handle_answer(y,Element):- assertz(Element).
handle_answer(n,Element):- assertz(reject(Element)).
handle_answer(u,Element):- assertz(undecided(Element)).

write_text([H]):- !,write(H),write('.').
write_text([H|T]):- write(H),tab(1),write_text(T).
```

Part B - model-building rule-book

```
possible(required, comp(C), ['create the compartment',C]):-
  output(C,S).
possible(required, comp(C), ['create the compartment',C]):-
  (flow(_,C,_);flow(_,_,C)).
possible(suggested, flow(grazing,C1,C2), ['add grazing flow between',C1,'and',C2]):-
  plant(C1), comp(C2), herbivore(C2).
```

Part C - ecological knowledge-base

```
plant(grass).
eats(rabbit,grass).
herbivore(X):- eats(X,Y), plant(Y).
```

Part D - modelling objectives

```
output(rabbit,biomass).
```

Fig. 2. Rule-based model-building program. Parts in bold are discussed in text; rest of program is included for completeness.

```
?- build_model
I will now create the compartment rabbit.
                                    (from first 'possible' rule)
I suggest that you add grazing flow between grass and rabbit.
                                    (from third 'possible' rule)
OK? y.=yes. n.=no, u.=undecided
y.
I will now create the compartment grass.
                                    (from second 'possible' rule)

comp(rabbit).
comp(grass).
flow(grazing, grass, rabbit).

Yes
```

Fig. 3. Transcript of a session with the rule-based model-building program. Parts in bold were typed by the user; the rest by the computer.

Part A – Model-building shell

This part is independent of the modelling domain; indeed, it could be used to help build other structures, such as the plan for a building. The core of this shell is the first clause for the predicate *build_model.*

Upon entering the Prolog query:

?–build_model.

the program attempts to find an action level (e.g. a 'required' or 'suggested' action), starting off with required actions, and only backtracking to suggested actions if there are no required actions outstanding. It searches through the set of model-building rules, attempting to find some possible element to include at that action level (e.g. add a *comp* or *flow* statement to the growing model blueprint): the *Text* it picks up will be used to generate an appropriate statement or question for the user. It then checks that this element has not already been included in the model, or rejected by the user. Finally, *handle* adds the current model element to the blueprint, if it is a required action, or asks the user if he wants it added, if it is suggested. The rule then calls itself – it is recursive – so that model elements can continue to be added until the blueprint is complete, when the first *build_model* rule fails. The second *build_model* rule then uses the built-in predicate *listing* to print the *comp* and *flow* clauses in the model blueprint.

Part B – Model-building rules

These rules specify the conditions that must be satisfied for a particular element to be added to the model blueprint.

The first argument of *possible* indicates whether the addition of the element specified in the second argument is required or suggested if the conditions are satisfied. The third argument is a list of canned text and variables, which together will be used to report to the user or formulate an appropriate suggestion. For example, the first rule says that it is required to create a compartment C IF a required output of the model is the amount of some substance *S* in some compartment *C*. The second rule ensures that a *comp(C)* assertion is added if there is a flow *from* compartment *C*, or a flow *to* compartment *C*. In the third rule it is suggested that an element specifying a *grazing* flow between compartments *C1* and *C2* be included IF *C1* represents some plant, and *C2* is a compartment which represents some herbivore. In this case, the text of the question that the user sees (using underlining and spaces to show how the phrase is built up) would be:

I suggest that you   *add grazing flow between*   *grass*   *and*   *rabbit*

Part C – Ecological knowledge base

This contains facts and rules relevant to the modelling domain, as previously discussed in Section 3.2.

Part D – Initial information

In order to provide direction to the modelling process, it is necessary that the user should be able to specify:
- objectives:"I am interested in plotting rabbit biomass against time";
- context:"The area I am thinking of is the uplands of Scotland"; and/or
- components and relationships that should be included: "The model should include grass and foxes".

In this simple example, the objective of having rabbit biomass as an output is the only thing specified, and that is done by a direct assertion to the Prolog database:

output(rabbit, biomass).

In practice, a simple question-and-answer program could pick up this information (albeit in a very stylised way) before proceeding to the model-building phase.

Apart from the minuteness of the rule-book and the ecological knowledge-base, the program is also very crude, and could readily be improved in a number of ways, in addition to the ways mentioned above. First, no

mechanism is included to allow parts of the model blueprint to be removed or modified. Second, as it stands, particular model-building rules either succeed or fail – no mechanism is provided which would generate questions as the program worked through the model-building rules and the ecological knowledge base. For example, the program should ask the user "Is rabbit an animal?" if this were not already known. Third, the choice of which model-building rule to try next should not depend on the arbitrary sequence of these rules in the program (as it does now), but should reflect a set of principles about how models should be constructed:
- adopt a top-down approach;
- tackle first those parts which are near to the stated goals rather than those further away; and
- tackle parts which are near parts recently handled.

4.3. *Improving the computational efficiency of the running of the model*

We have already seen how 'simulating the behaviour of the model' can be handled using the standard Prolog query mechanism. The main reason for the problems with that approach is that (unlike conventional programming languages) intermediate values are not stored during the course of evaluation in such a way that they can be accessed: they are only held internally by Prolog during the course of recursion. In order to run the simulation more efficiently, we can adopt one of three possible approaches:

(1) Add a fact stating the current value of a variable each time it is determined. Prolog provides a built-in predicate *assert* which can be used to add *value* facts to the program. However, the subgoal:

..., *assert(value(Variable, Value, Time)).*

would need to be added to *every* value rule for intermediate, rate and state variables. This is a messy solution, since it destroys the declarative nature of the *value* predicate, and in fact is not complete, since it would also be necessary to check that the value has not already been asserted.

(2) Extend the standard Prolog interpreter, to make it more suitable for model simulation by automatically storing the current value when evaluating *value* rules. Having done this, we need to make no changes to the blueprint specification: all the rules remain the same, and we can ask the same questions as before about the future state of the system. The difference is that the simulation is much more efficient, since the immediately preceding values can be found directly.

(3) Do not use Prolog for simulation, but generate a program in (say) Fortran or a simulation language. This is in fact straightforward for system

dynamics models, since they have such a well-defined structure, and the methods we use are very similar to those described in Section 4.1 for producing descriptions of the model: the main extra requirement is to ensure that the influence functions are output in the correct order for evaluation.

5. CONCLUSIONS

The approaches discussed in this paper are directly relevant to the three concepts presented in the Introduction.

First, we have shown that Prolog offers a powerful formalism for representing ecological simulation models conforming to the system dynamics formalism. The representation includes the symbolic, conceptual aspects of a model, as well as the mathematical and computational aspects. This means that the complete model structure, at all levels, is 'explicit' and 'transparent': other people can investigate, evaluate and compare model structures in a way that is not feasible with a textual description or an implementation as a conventional program. A logical consequence of this approach is the development of a 'model library', containing many hundreds of model blueprints: searches can then be made for all models sharing some set of features.

Second, we have shown that some aspects of modelling knowledge – for example, rules for detecting errors in models, and rules about building models – are also naturally represented in Prolog. The representation of modelling knowledge can thus be integrated with the representation of model structure, enabling the rules to be used in practice to check models and to offer guidance on their construction. Perhaps more important, however, is the fact that the rules are an 'explicit' statement of modelling knowledge: modellers should feel encouraged (obliged?) to justify the decisions made in constructing a particular model by reference to such rules. Indeed, one could argue that the ecological modelling literature should concentrate on the dissemination of such knowledge, rather than on the mere description of particular models. While authors frequently state their assumptions, it is much more valuable to know why these assumptions were made.

Third, we have shown that it is feasible to devise programs that can guide ecologists through the process of constructing a model to satisfy some objective. By opening up the modelling process to all ecologists, rather than just those who happen to possess the necessary modelling and programming skills, no ecologist should be prevented from adopting a systems approach when that is suitable and relevant, and all ecologists should be in a better position to question the assumptions of other people's models.

We have taken the unusual approach of describing how these various facets of modelling can be tackled using a particular language – Prolog –

rather than of describing a particular software system that has the same features. We have done this in order to stimulate modellers to try out these ideas for themselves, and thus to encourage informed discussion of two key issues. The first, assuming that the 'blueprint' concept is adopted, is the development of a standard formalism for representing system dynamics models – i.e. an agreed set of Prolog predicates – taking into account considerations such as database design. The second issue is the representation of modelling elements which do no fit into the system dynamics formalism, such as complex substructure and multiple attributes per object. Work undertaken in the Edinburgh ECO project has confirmed the expressive power of Prolog for representing such elements. In addition, we have implemented a model-building environment (the EcoLogic system (Robertson et al., 1988)) which uses an even more powerful ('sorted') logic as the knowledge representation language.

Ultimately, however, the really challenging problems are not those concerned with model representation. Rather, they are concerned with the conceptual aspects of ecological modelling: the types of knowledge that ecological modellers possess; the strategies that modellers use for applying that knowledge; the relationship between an ecologist's objectives and the resulting model; and bridging the conceptual gap between ecologist and computer (Bundy, 1984). There are many difficult issues here, and it is vital that simplistic, short-term solutions should be seen as merely the first few steps on a long journey. However, the goal is worth pursuing: answering these questions will not only open the door for software systems that improve the accessibility of modelling, but will also yield great insight into the modelling process, and therefore increase its rigour.

ACKNOWLEDGEMENTS

The Edinburgh ECO project is supported by SERC grant GR/E/00730.

APPENDIX

Simple menu-based model display program

```
display:-
    write('1 - List all compartments'), nl,
    write('2 - List all flows for one compartment'), nl,
    read(Choice),
    display(Choice).
```

```
display(1):-
   write('The compartments in the model are:'), nl,
   comp(C),
   write(C), nl,
   fail.
display(1).

display(2):-
   write('Compartment:'), nl,
   read(C),
   write('Inflows:'), nl,
   display_inflows(C),
   write('Outflows:'), nl,
   display_outflows(C).

display inflows(C):-
   flow(F, C1, C),
   write(['Flow  ', F, '  from  ', C1]), nl,
   fail.
display_inflows(_).

display_outflows(C):-
   flow(F, C, C1),
   write(['Flow  ', F, '  to  ', C1]), nl,
   fail.
display_outflows(_).
```

This program makes use of 'failure-driven loops' to find all possible solutions to some query. The use of the built-in predicate *fail* forces the Prolog interpreter to look for another solution – but in the meantime it has written out the current solution using the built-in predicate *write*. This programming approach is quick to implement, but it is not very elegant. A better method is to make use of predicates that find all possible solutions to a goal – most Prolog interpreters provide predicates such as *bagof* or *setof* which perform this task.

REFERENCES

Bratko, I., 1986. Prolog Programming for Artificial Intelligence. Addison-Wesley, Reading, MA, 423 pp.

Bundy, A., 1984. Intelligent Front Ends. In: M.A. Bramer (Editor), Research and Development in Expert Systems. Cambridge University Press, Cambridge, pp. 193–204.

Burnham, W.D. and Hall, A.R., 1985. Prolog Programming and Applications. Macmillan, London, 114 pp.

Fjellheim, R.A., 1986. A knowledge-based interface to process simulation. In: E.J.H. Kerckhoffs, G.C. Vansteenkiste and B.P. Zeigler (Editors), AI Applied to Simulation. Simulation Series, 18 (1). Society for Computer Simulation (Simulation councils, Inc), San Diego, CA, pp. 97–105.

Fujiwara, R. and Sakaguchi, T., 1986. A expert system for power system planning. In: E.J.H. Kerckhoffs, G.C. Vansteenkiste and B.P. Zeigler (Editors), AI Applied to Simulation. Simulation Series, 18 (1). Society for Computer Simulation (Simulation Councils, Inc), San Diego, CA, pp. 174–177.

Leaning, M.S. and Nicolosi, E., 1986. MODEL – software for knowledge-based modelling of compartmental systems. Biomed. Meas. Inf. Control, 1: 171–181.

Lewis, J., 1986. STELLA: a model of its kind. Pract. Comput., 9(9): 66–67.

Loehle, C., 1987. Applying artificial intelligence techniques to ecological modelling. Ecol. Modelling, 38: 191–212.

Malpas, J., 1987. Prolog: A Relational Language and its Applications. Prentice-Hall, London, 464 pp.

Muetzelfeldt, R., Uschold, M., Bundy, A., Harding, N. and Robertson, D., 1986. ECO: an intelligent front end for ecological modelling. In: E.J.H. Kerckhoffs, G.C. Vansteenkiste and B.P. Zeigler (Editors), AI Applied to Simulation. Simulation Series, 18 (1). Society for Computer Simulation (Simulation Councils, Inc), San Diego, CA, pp. 67–70.

Muetzelfeldt, R., Robertson, D., Uschold, M. and Bundy, A., 1987. Computer-aided construction of ecological simulation models. In: C.A. Kulikowski, R.M. Huber and G.A. Ferrate (Editors), Artificial Intelligence, Expert Systems and Languages in Modelling and Simulation. Elsevier, Amsterdam, pp. 385–393.

Robertson, D., Muetzelfeldt, R., Plummer, D., Uschold, M. and Bundy, A., 1985. The ECO browser. In: Expert Systems 85. British Computer Society Specialist Group on Expert Systems, Coventry, pp. 143–156.

Robertson, D., Bundy, A., Uschold, M. and Muetzelfeldt, R., 1988. Using ecological descriptions to guide the construction of simulation programs. Res. Pap. 381, Department of Artificial Intelligence, University of Edinburgh, 4 pp.

Rogers, J.B., 1986. A Prolog Primer. Addison-Wesley, Reading, MA, 219 pp.

Sterling, L. and Shapiro, E., 1986. The Art of Prolog, MIT Press Series in Logic Programming, Massachusetts Institute of Technology Press, Cambridge, MA, 427 pp.

Uschold, M., Harding, N., Muetzelfeldt, R. and Bundy, A., 1985. An intelligent front end for ecological modelling. In: T. O'Shea (Editor), Advances in Artificial Intelligence. Proc. 6th Eur. Conf. ECAI-84, 5–7 September 1984, Pisa. North-Holland, Amsterdam, pp. 13–22.

Wolfe, I.R., Zweig, R.D. and Engstrom, D.G., 1986. A computer simulation model of the solar–algae pond ecosystem. Ecol. Modelling, 34:1–59.