TITLE:
Modelling ecological interaction despite object-oriented modularity

AUTHOR:
J. F. Derry

ADDRESS:
Institute of Cell, Animal and Population Biology,
University of Edinburgh,
West Mains Road,
Edinburgh,
EH9 3JT,
Scotland, UK

EMAIL:
Julian.Derry@ed.ac.uk

CORRESPONDANCE ADDRESS:
as above

published as

Derry, J.F. (1998)
Modelling ecological interaction despite object-oriented modularity.
Ecological Modelling, 107, 145-158.

Abstract

Ecological modellers may be deterred from using object-oriented programming languages by reports of the inherent rigidity of the object-oriented formulation, giving rise to a poor representation of biotic interaction. *Internal* approaches of imitating relationships between the components of an ecological model bury the associated code inside the object declarations of one or more of these components. *External* approaches isolate a component's processes thereby maintaining the object's integrity, but at the cost of realism. Highlighted here is a frame-based technique that uses object pointers to reference the active components thereby allowing mimicry of interaction while maintaining individual object integrity.

Keywords

object-oriented, frame-based, state-and-transistion, interaction, super-individuals, semi-arid grazing system

1. Introduction

Guthery (1989) warned that "Object-oriented programming makes building code fragments easier, but it makes integration much more difficult. Making the easy parts easier but the hard parts harder is not progress...". For the ecological modeller, integration involves generation of an analogy for the, often complex, biotic interactions between the components of a natural system (Maley and Caswell, 1993). Ecological modellers have found features of the object-oriented paradigm (OOP) attractive for representation of these system components. However, the same features have been found inflexible when applied to the interactions that determine the dynamics of such systems (Silvert, 1993). Computer scientists may know how to overcome these problems, but ecologists usually do not. This paper sets out to highlight a technique that facilitates integration in object-oriented models and illustrate its application in an ecological context.

At the level of the computer program that may be used to implement a model, data is stored in *variables* used to characterise each of the participating components. An interaction is simply a calculation carried out on that data. Structured programming languages (e.g., C and Pascal) require that a variable's datum can only be used in a calculation if it is visible within the region of the program that contains its point of use. A datum's visibility within this region is termed its *lexical scope* (Kernighan and Ritchie, 1988). Invisibility of a variable, when it is deemed out of lexical scope, has arisen from the development of structured programming languages. Traditional programming languages (e.g., BASIC and Fortran) are not structured and therefore tend to avoid problems associated with lexical scope by giving their data global scope. All variables are visible at any point in the program. At the other

extreme, object-oriented programming languages (e.g., C++ and object-oriented Pascal) tend to package data into *objects*, each object defined by a *class*, the properties of which may be inherited by subsequent classes further down a class inheritance tree. Lexical scope is limited to the bounds of this object hierarchy allowing the generation of complex system components by the superimposition of simpler constituent parts. The result is that the hierarchical and modular nature of OOP has been widely adopted for its suitability to represent system components (Webster, 1994). For the same reasons, these properties of OOP also negate the flexibility required to model the biotic interaction when applied to ecological systems (Silvert, 1993).

OOP's inherent structure has resulted in the development of three distinct approaches to integration. For convenience, the approaches are termed in this paper, *internal*, *external* and *intermediate*. Amongst the OOP ecological models that have been reported in the literature, there are modellers who have favoured a convention of nesting the calculations denoting interaction within the objects representing one or more of the participating elements (e.g., Silvert, 1993; Durnota, 1996). Silvert (1993) presents an OOP implementation of a Lotka-Volterra model to demonstrate how this *internal* approach may be used to feed a plankton object to a fish object. To do so, a copy of the prey object is passed to the predator object where a proportion of the prey biomass is then allocated for consumption. A more complex interaction, for example one that involves more participants or is the cause of subsequent interaction, will require careful synchronisation and correctly parameterised repetition of this process. Therefore, in terms of simulation time, computer resources and code efficiency, this style favours representation of simple interactions between two objects.

To facilitate the modelling of more complicated relationships, an *external* approach would be to generate a distinct block of code to represent processes

involving more than one object. The advantage here is the isolation of processes solely akin to an object, thereby maintaining that object's integrity. This is a typical technique for extending the *internal* approach model. Indeed, Silvert (1993) adopts this approach to extend his example application of a predator species of fish and a single prey of plankton to a larger menu which includes more prey items, orchestrated at each iteration by a distinct coding block called "Integrate". The *external* approach is often convenient for the programmer, but the conceptual pitfall is that interaction is considered as a separate entity rather than the union of object properties.

The approach described in this paper is an *intermediate* alternative, where an object is declared with links to the properties of another object with which we may expect it to interact, thereby predefining the range and type of interaction (Maley and Caswell, 1993). Object integrity is thereby maintained, while allowing interactions between the properties of two or more objects, and is closer to what happens in the real world. Individuals and populations maintain their integrity regardless of interaction and only the participants are actively contributing to the effects of the interaction on the system at any one time.

Parallels may be made between this approach and *frame-based* or *state-and-transition* approaches to ecological systems modelling. Each combination of actively participating components defines a state for the model, from which it may pass into another state upon change in the interaction between its components. Starfield et al. (1993) developed a frame-based approach to depict a state-dependent mechanism for *Brachystegia* woodland dynamics. The system dynamics arising from the combinations of frame choice and the switching between those combinations is compared to those of a state-and-transition model (Westoby et al., 1989). However, it should be noted that unlike a transition, the process of decoupling one frame and the

coupling of another cannot be considered a state in itself. Although Starfield et al. (1993) described their modelling paradigm in very object-oriented terms, the model was not implemented using OOP (D. H. M. Cumming, pers. comm.).

In summary, it is considered that the modularity of OOP is convenient for the packaging of information organised into categories of kind. Using the integrity that this characteristic confers on a component type, and employing an approach that incorporates interaction between the components of a system by the use of couplings analogous to a frame-based approach, it is possible to maintain the isolation and separate operation of processes associated with such a type, whilst allowing the integration of components by the generation of states for the model. Each state is the result of a combination of interactions between two or more system components, and each interaction is made possible at the programming level by the existence of remote links between potential associates. This paper presents the main components of an aspect of a wide-reaching and well-documented ecological system and their interaction as an example of how links may be manufactured between the modules of an object-oriented ecological model in an attempt to improve the accuracy with which it is possible to represent naturally occurring relationships.

2. Method

The steps required to create object-oriented programs are already satisfactorily documented and it is not intended to repeat basic information here. Only the relevant aspects of object-oriented programming languages are considered to facilitate comprehension of the approach to modelling OOP implemented interaction. Code

examples are presented in C++ (Borland, 1991), however, the syntax is comparable to other OOP languages. For clarity, C++ keywords are shown in bold, and, variable and class names in italics.

2.1. *Brief description of the model*

The dynamics of a semi-arid grazing system may be simulated by the coupling of animal population dynamics to the plant biomass dynamics (Derry et al., in prep. A recent discussion of the nature of this coupling may be found in Illius and Gordon, 1998). Classes detailing pertinent aspects of each component were incorporated into distinct modules. Fig.1 shows an overview of the interactions between the animal and plant modules in the model.

insert Fig.1

Initially, these interactions appear simple. However, upon inclusion of detail; constituent plant parts and their selection and intake by animals, the interactions between the components become more complex, and the modelling of them is exacerbated by increasing the diversity of plant and animal species that need to be included in the model.

2.2 *Iteration of the model*

Daily rainfall data is used to generate soil water levels after run-off, infiltration and drainage. Daily growth for each species present within the plant module is predicted from the soil water level. Allocation of plant growth is based on the morphology of each vegetation type in combination with its phenological rates. Herbivore diet selection and consumption of the forage is directed by the specific metabolic requirements of each animal species present. This is modelled as the metabolic requirement for maintenance on an allometric basis, calculated with respect to the constraints of forage quantity and quality. A more detailed description of the model can be found in Derry et al. (in prep.).

The single interaction in the model that incorporates the greatest range of possible combinations is herbivore diet selection. We will now focus on the programming that contributes to the representation of this interaction in the model as it will best illustrate application of the *intermediate* approach. Daily diet selection is calculated as the single species of forage that has the greatest potential energy intake after the costs of foraging have been taken into account, according to the energy maximisation premise of optimal foraging theory (Stephens and Krebs, 1986). Diet selection is dependent on the allometry of each animal species, therefore, selection of a purely herbaceous diet, as would be expected for a grazer, or selection of a woody diet, as would be expected for a browser, will be largely determined by differences in body size. A mixed diet feeder might be expected to select forage from both herbaceous and woody types, and this would be possible without alteration of the programming code that comprises the interaction mechanism.

2.3. *Code implementation*

The diversity of plant and animal species required to model a semi-arid grazing system necessitated a program structure that would generate generic vegetation and herbivore types. Subsequent parameterisation was then used to differentiate between species within each module in order to generate a single object for each population comprising an individual plant or animal species.

2.3.1. *Making modules*

The construction of object-oriented ecological models requires the declaration of classes within modules and the linking of those modules. Each module may contain a set of classes that are akin to one another, providing a mechanism for the ecological modeller to group data (*members*) and calculations (*methods*) in terms of taxonomy (e.g., Makela et al., 1993), scale (e.g., Righter and Ford., 1994) or function (e.g., Sequeira et al., 1993). Classes can therefore be used as abstract definitions of a biological type and the placing of these classes within a hierarchical structure allows the inheritance of a parent class's properties by its descendants. The lexical scope of members and methods down the hierarchy is achieved by labelling them **public** (in contrast to **private** which restricts their visibility to that class).

2.3.1.1. *Plant module*

The morphological differences between vegetation types required a more expanded structure than that seen below for the animal module. Each vegetation type; annual grass, perennial grass, forbs, shrubs and trees, was deemed either herbaceous or woody in nature and assigned characteristic "plant parts" categories to reflect the differences in morphology (Appendix A). Members and methods common to all vegetation types were declared at the highest levels of the module's hierarchy (Table.1a). Complexity was generated by inheritence of characteristics down this hierarchy (Table.1b). *In vitro* digestibilities of edible plant parts were included (Table.1c). For herbaceous types these were green leaf, dead leaf, green stem and dead stem, and for woody types these were green leaf, dead leaf, green stem and fallen seed.

insert Table.1

Morphological differentiation was achieved by branching of the hierarchical tree at lower levels (Fig.2). Plant species of the same vegetation type were grouped in a dynamic array, a data structure flexible in size so as to cater for the desired number of species.

insert Fig.2

2.3.1.2. *Animal module*

Impact from herbivory differs with respect to an individual's age and reproductive state. That individual's contribution to the overall stocking rate is also dependent on it's age and reproductive state. To accommodate this animal populations were categorised into sex and age classes (Fig.3), so the allometric relationship between forage intake and body size allowed treatment of consumption estimates at a higher resolution than possible at the animal population level (Appendix B).

insert Fig.3

Calculations were not made for individuals but for *super-individual* age classes (Scheffer et al., 1995) using the class mean values for animal state (*FFMASS* and *FAT*) to represent all individuals therein. Animal stocking rate was thus calculated by the use of the livestock equivalent (LE) of an age class, where 1 LE is the $MASS^{0.75}$ of a mature bull of 450 kg. Therefore, the overall stocking rate may be defined in terms of the cumulative effect of the LE of each animal category.

2.3.2. *Integration*

Programming languages use *data types* to specify which operations may be carried out on the members being stored. For example, the division of a word is as meaningless as the concatenation of two decimal numbers, whereas, searching "hello" for the letter "e", or the subtraction of one integer from another are valid operations.

Calculations comprise sequences of one or more operations.

By declaring a class we are also defining a data type. Admittedly, a class may contain more than just members; it is also somewhere to locate the code for its methods. However, it is convenient to think of a class in terms of a data type as, like regular data declaration, it is also a request to the computer for allocation of memory for storage of its contents. When the OOP program is running, its objects are simply the packages of computer memory allocated for the members and methods defined by its classes, a process called *instantiation*.

2.3.2.1. *Declaring potential interaction*

At run-time, computer memory is referenced by the use of *pointers*. Objects and data are stored in sections of memory at specific numeric locations called *memory addresses*, and a pointer is the starting address for a section of memory. As classes are data types, the objects that result from them can also have pointers targeted at their starting memory addresses (Table.2).

insert Table.2

Provided that the lexical scope of a class declaration is accounted for, object pointers may be declared within the class declaration of other objects (Table.3).

insert Table.3

The level at which interaction may be represented is therefore defined by the location of these pointer declarations.

2.3.3 *Interaction*

Diet selection was calculated once per iteration for each animal species, but defoliation was allometrically derived for each selected plant part in terms of each animal age class. To calculate defoliation from a selected forage species, object pointers were declared for each vegetation type within the age class declarations for the animal module. At the outset the object pointers were created empty (null or equal to zero). To represent selection of a diet, and activation of the interaction between the animal species and a plant species, the object pointer is assigned a plant object's starting memory address (Table.4), otherwise it remained empty.

insert Table.4

The pointer supplies a link between interacting objects, but as mentioned above, it is not the component objects that interact directly but a property of each object that does the interacting. An object's properties are stored in the members associated with a class and they can be accessed using a *member selector*. Member selectors can either

be used for direct or indirect access (Table.5).

insert Table.5

Indirect access to the plant parts was allowed by use of their member selectors. To deactivate the interaction with an object, the object pointer was nullified. In this way it is possible to generate interaction states between an unlimited number of components, for an unlimited amount of simulation time. Implementation of the code is simple and non-representative of the complexity of interaction that may be attained (Fig.4).

insert Fig.4

Because the object pointers are declared within a class declaration that is inherited by all objects in the animal module, selection of a forage by one animal species does not preclude selection of that same forage by another animal species. Also, increasing the number of plant and animal species does not limit the number of interactions that may occur.

3. Tests of efficacy

Validation of the pointer mechanism *per se* is an exercise in computer science

and reproduction of these analyses would be contrary to the aim of this paper. In preference, presented here is a result that illustrates a benefit of using the *intermediate* approach when integrating the biotic components of the grazing ecosystem model described above. In this particular application, the diversity of animal and plant species in the natural system necessitated capacious structuring of the relevant modules. Abstraction, tolerated by object-oriented design, permitted the definition of generic types, whilst subsequent parameterisation differentiated objects generated by multiple instantiation of the same class. A large range of potential interactions were apparent. Simulations of goats and cattle mixed grazing trials were used to test whether the *intermediate* approach, comprising object pointers and member selectors, was capable of this variety.

Diet selection and intake for each species of herbivore was predicted daily by an allometric model (Illius and Gordon, 1998). For the correct operation of object pointers coupling super-individuals to their diet, simulation output should reflect variation in calculated intake due to differences between and within animal species (Fig.5a, b), and variation in the proportions of plant parts removed (Fig.5c), as determined by predictions of diet selection (Illius and Gordon, 1998).

4. Conclusions

The ecological modeller is faced with the problem of identifying a suitable computational tool. OOP has proved itself beneficial for the representation of the components of a system because of the modular structure that it confers on a modelling program, however, it has been criticised for its lack of flexibility that may

be required to emulate biological interaction. Presented here is a approach that uses the modular characteristic of OOP to provide a means to manipulate object-oriented code so that the relationships between the objects imitate better the interaction dynamics of the natural system being modelled.

The need for improved integration may be questioned; what does it matter which style is used; *internal*, *external* or *intermediate*, as long as it proves efficacious? It matters when the adopted style is inhibitory to progress, or may be modified for efficiency. The need for improved integration has been already reached by consensus; "...the object-oriented paradigm should be seen not only as just another way of programming but as a more expressive way of naturally representing a problem...", (Gherman and Hessman, 1995). It now remains to develop better ways in which to implement OOP as an ecological modelling tool.

The object pointer is a standard feature of OOP programming and can be anchored between the objects representing system components in a way that tends towards the relationships that exist between components of a natural system. Object pointers can be used to either; declare a range of interactions, as would be expected for explorers, (e.g., foragers), or, declare an exact location if the target object is already known (e.g., host-specific parasites). The resolution can be fitted to the scale and nature of the model.

Finally, the capacity of the object pointer approach is not restricted by the number of system components, allowing a realisation of the maximum number of potential interactions for natural systems in which all combinations of the components are involved.

Ways are being found to improve the application of OOP in ecological modelling. Some are already well known techniques within computer science,

however, the training of biologists often precludes them from this knowledge base. Papers such as this will highlight techniques that are already known or are yet to be developed.

Acknowledgements

I would like to thank the following for their critical review of the first draft of this manuscript; R. Muetzelfeldt, L. Partridge and N. Outram. A. Illius and I. Gordon engaged in valuable discussion. Two anonymous referees provided further useful comments. This work was carried out whilst funded by the Overseas Development Administration (NRI X0289).

References

Borland, 1991. Turbo C++ 3.1 For Windows: Programmer's Guide. Borland International Inc., 389 pp.

Capper, D. M., 1994. Introducing C++ For Scientists, Engineers and Mathematicians. Springer-Verlag, 502 pp.

Durnota, B., 1996. An abstract object model of an animal's environment. Ecol. Modelling, 86: 119-123.

Gherman, D. C. and Hessman, F. V., 1995. Object-oriented computing in the natural sciences. IEEE Computational Science & Engineering, 2(1): 90-91.

Guthery, S., 1989. Are the emperor's new clothes object-oriented? Dr. Dobb's

Journal, Dec. 1989: 80.

Guthery, S., 1992. A curmudgery on programming language trends. Dr. Dobb's Journal, Dec. 1992: 104.

Illius, A. W. and Gordon, I. J., 1998. Scaling up from functional response to numerical response in vertebrate herbivores. In: H. Olff, V. K. Brown and R. H Drent (Editors), Herbivores, plants and predators, Blackwell Science, Oxford. (in press).

Kernighan, B. W. and Ritchie, D. M., 1988. The C Programming Language. 2nd edition. Prentice Hall Software Series, USA, 272 pp.

Makela, M. E., Rowell, G. A., Sames IV, W. J. and Wilson, L. T., 1993. An object-oriented intracolonial and population level model of honey bees based on behaviors of European and Africanized subspecies. Ecol. Modelling, 67: 259-284.

Maley, C. C. and Caswell, H., 1993. Implementing *i*-state configuration models for population dynamics: an object-oriented approach. Ecol. Modelling, 68: 75-89.

Righter, R. and Ford, R., 1994. An object-oriented characterization of spatial ecosystem information. Mathl. Comput. Modelling, 20(8): 17-29.

Scheffer, M., Baveco, J. M., DeAngelis, D. L., Rose, K. A. and van Nes, E. H., 1995. Super-individuals a simple solution for modelling large populations on an individual basis. Ecol. Modelling, 80: 161-170.

Sequeira, R. A., Stone, N. D., Makela, M. E., El-Zik, K. M. and Sharpe, P. J. H., 1993. Generation of mechanistic variability in a process-based object-oriented plant model. Ecol. Modelling, 67: 285-306.

Silvert, W., 1993. Object-oriented ecosystem modelling. Ecol. Modelling, 68: 75-89.

Starfield, A. M., Cumming, D. H. M., Taylor, R. D. and Quadling, M. S., 1993. A

frame-based paradigm for dynamic ecosystem modelling. AI Applications, 7 (2/3): 1-13.

Stephens, D. W. and Krebs, J. R., 1986. Foraging theory. Princeton University Press, New Jersey, 247 pp.

Webster, S., 1994. An annotated bibliography for object-oriented analysis and design. Information and Software Technology, 36(9): 569-582.

Westoby, M., Walker, B. H. and Noy-Meir, I., 1989. Opportunistic management for rangelands not at equilibrium. J. Range Manage, 42(4): 266-274.

Appendix A

Plant module class declarations.

insert Table.6

Appendix B

Animal module class declarations.

insert Table.7

Table.1a

```
class Plant                    {      // begin class declaration
                ...
              public:                 // lexical scope
                ...
                float gleaf ;         // member declarations
                float dleaf ;
                float gstem ;
                ...
              };                      // end class declaration
```

A simplified syntax for the declaration of the *Plant* class in the plant module. *Plant* is at the highest level of the module's hierachy and contains some members (green leaf : *gleaf*, dead leaf : *dleaf* and green stem : *gstem*) that will be common to all vegetation types. The label **public** determines that the lexical scope of these members will be maintained in the descendents. **float** is a decimal data type. Semi-colons are used to mark the end of a line. Code may be annotated by the inclusion of comments preceded by a double forward slash (//) which informs the compiler software to ignore the comment.

Table.1b

```
class Vegetation : public Plant {       // begin class declaration
...
};                                      // end class declaration
```

A simplified syntax for the declaration of the *Vegetation* class in the plant module. *Vegetation* inherits all of the members and methods previously declared **public** in *Plant*.

Table.1c

```
class Forage : public Vegetation {    // begin class declaration
  ...
 public:                              // lexical scope
  ...
  float gldig ;
  float dldig ;
  float gsdig ;
  ...
 };                                   // end class declaration
```

A simplified syntax for the declaration of the *Forage* class in the plant module. *Forage* inherits all of the members and methods previously declared **public** in *Plant* and in *Vegetation*, whilst adding the *Plant*'s plant parts forage characteristics, (green leaf digestibility : *gldig*, dead leaf digestibility : *dldig* and green stem digestibility : *gsdig*).

Table.2

AnnualGrass PointerToAnnualGrass* ;

Declaration of an object pointer. A pointer is denoted by the use of an asterisk (*). Therefore, the pointer, *PointerToAnnualGrass*, is declared as being of the data type *AnnualGrass**, the starting memory address of an object defined by the class *AnnualGrass*.

Table.3

```
class AnnualGrass : public Herbaceous     {     // begin class declaration
                                ...
                              } ;                 // end class declaration

class Age                                 {     // begin class declaration
                                ...
                                AnnualGrass*  PointerToAnnualGrass ;
                                                  // pointer declaration
                                ...
                              } ;                 // end class declaration
```

Declaration of potential interaction. The pointer declaration for *PointerToAnnualGrass* within the class declaration for *Age* is made possible by previous inclusion of the class declaration for *AnnualGrass*. This ensures that the data type *AnnualGrass** will be in lexical scope.

Table.4

AnnualGrass Aristida ;

PointerToAnnualGrass = &*Aristida* ;

Definition of an object pointer. *Aristida* is an object of the data type defined by the class *AnnualGrass*. The referencing operator (&) returns the *Aristida* object's starting address and stores it in the object pointer *PointerToAnnualGrass*.

Table.5

Herbivore Cow ;

Cow.Male.Age[*OldestAge*].*HerbaceousConsumption* =

Cow.Male.Age[*OldestAge*].*PointerToAnnualGrass* →

gleaf

+

Cow.Male.Age[*OldestAge*].*PointerToAnnualGrass* →

gstem ;

Access of an object property via its class member selector. The dot (.) is used for direct access and the right arrow (→) for indirect access via an object pointer.

Table.6

Class	Use	Member	Description
Plant			generic plant type
	structure	*gleaf*	green leaf, kg/ha
		dleaf	dead leaf, kg/ha
		gstem	green stem, kg/ha
		seed	inflorescence, kg/ha
		stores	carbohydrate stores used for growth, kg/ha
		fseed	fallen seed, kg/ha
		germ	germinating seed, kg/ha
	water balance	*acttr*	transpiration, mm/kg/ha
		aevap	evapotranspiration, mm/kg/ha
	growth	*gwth*	daily growth, kg/ha
	phenology	*sstart*	season start date, days from September 1st.
		send	season finishing date, days from September 1st.
Vegetation			generic vegetation type
	identification	*UserType*	user-definable name tag for each species
		ApplType	application name tag for each species
Forage			generic forage type
		gldig	digestibility of green leaf, *In vitro* DMD
		dldig	digestibility of dead leaf, *In vitro* DMD
		gsdig	digestibility of green stem, *In vitro* DMD
		consumed	daily consumption by animals, kg/ha
Herbaceous			generic herbaceous plant type
		dstem	dead stem, kg/ha
		dsdig	digestibility of dead stem, *In vitro* DMD
AnnualGrass			generic annual grass type
ArrayOfAnnualGrasses			container for annual grass species
PerennialGrass			generic perennial grass type
ArrayOfPerennialGrasses			container for perennial grass species
Forb			generic forb type
ArrayOfForbs			container for forb species
Woody			generic woody plant type
		wood	wood, kg/ha
		fsdig	digestibility of fallen seed, *In vitro* DMD
		carea	canopy area, ha
		popsize	number of individuals in the population
		treemass	total biomass of individuals in the population, kg/ha
Shrub			generic shrub type
ArrayOfShrubs			container for shrub species
Tree			generic tree type
ArrayOfTrees			container for tree species

Table.7

Class	Use	Member	Description
Neonate			non-sexed neonates
	sex	*male*	male proportion of births
		female	female proportion of births
Age			declaration of a sexed age class
	population	*heads*	total individuals in the age class
		biomass	total biomass of the age class, kg
		starved	annual total of mortalities by starvation
		disease	annual total of mortalities from disease
	demography	*RECRUITS*	temporal distribution
	state	*FAT*	mean fat mass, kg
		FFMASS	fat free mass, kg
		MASS	total mass, kg (= *FAT* + *FFMASS*)
	diet	*PointerToAnnualGrass*	pointer to an *AnnualGrass* object
		PointerToPerennialGrass	pointer to a *PerennialGrass* object
		PointerToForb	pointer to a *Forb* object
		PointerToShrub	pointer to a *Shrub* object
		PointerToTree	pointer to a *Tree* object
		HerbaceousConsumption	herbaceous mass in the daily selected diet, kg
		WoodyConsumption	woody mass in the daily selected diet, kg
Male			juvenile and adult males
Nonpregnant			juvenile and adult nonpregnant females
Pregnant			adult pregnant females
Lactating			adult lactating females
	reproduction	*AverageMilkYield*	average milk yield for the age class, kg
Female			juvenile and adult females
Herbivore			generic herbivore type
	identification	*UserType*	user-definable name tag for each species
		ApplType	application name tag for each species
	population	*heads*	total individuals in the species
		biomass	total biomass of the species, kg
		starved	annual total of mortalities by starvation
		disease	annual total of mortalities from disease
	demography	*YoungestAge*	first juvenile age class
		YoungestBreedingAge	first reproducing age class
		OldestAge	last age class
	reproduction	*gestation*	gestation time, days
		lactation	lactation time, days
	state	*MinRange*	minimum mean fat mass, kg
		MaxRange	maximum mean fat mass, kg
	metabolism	*avMM*	average multiples of maintenance from daily diet
	offtake sales	*requirement*	required offtake for applied management policy
		heads	number of individuals sold
		biomass	mass of the individuals sold, kg
	management	*AverageStockingRate*	actual stocking rate, LE/ha
		AdaptiveStockingRate	applied stocking rate for tracking policy, LE/ha
		CapStockingRate	maximum stocking rate for tracking policy, LE/ha
		ConstantStockingRate	maximum stocking rate for fixed policy, LE/ha
		TrackingMortality	tracking mortality policy option
		PanicThreshold	pre-emptive sales policy rainfall threshold, mm
ArrayOfHerbivores			container for animal species

Fig.1.

Interaction between the animal and plant modules in the model.

Fig.2.

Class inheritance diagram for the plant module. *Plant* is the *base class*, the class at the highest level of the hierachy. The higher levels contain declarations for members and methods common to all lower levels. For example, *Forage* is a 'kind of' *Vegetation* (Capper, 1994). Differentiation is achieved by splitting the descendancy. Plant species of the same vegetation type are grouped in a dynamic array, the size of which is determined by *n*, a user-defined parameter specified separately for each vegetation type.

Fig.3a.

Class inheritance diagram for the animal module. *Age* is the base class from which each age class derives its structure. Each population of herbivores is considered a *container* for the age class strata (Guthery, 1992). Classification as a neonate occurs for less than a year, the smallest unit of time used to define the age class structure. All animal species were grouped in the same dynamic array the size of which is determined by *n*, a user-defined parameter specifying how many animal species to include.

Fig.3b.

Structure resulting from the *Age* class inheritance for the classes M : *Male*, NP : *Nonpregnant*, P : *Pregnant*, and L : *Lactating*. *b* and *d* are user-defined parameters specifying, the youngest reproducing age class, and how many age classes to include in each sex, respectively, and are specified separately for each animal species.

Fig.4.

Schematic diagram showing the range of diet selection between two forage species (one herbaceous and one woody in nature) by three animal species within the model.

Fig.5.

The value of the *intermediate* approach may be illustrated by the range and detail of interaction that can be modelled. In this example, calculated intake for two species of herbivores, a) goats and b) cattle, is shown to vary across age classes due to differences in size and reproductive status. Member selectors couple each age class to components of the vegetation, such that, prediction of daily diet may lead to c) temporal variation in the removal of plant parts.

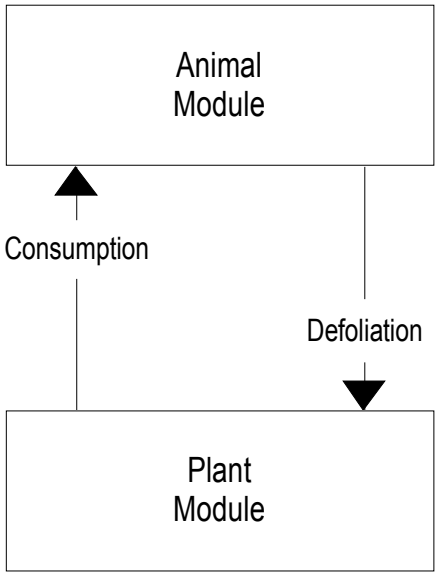


Fig.1

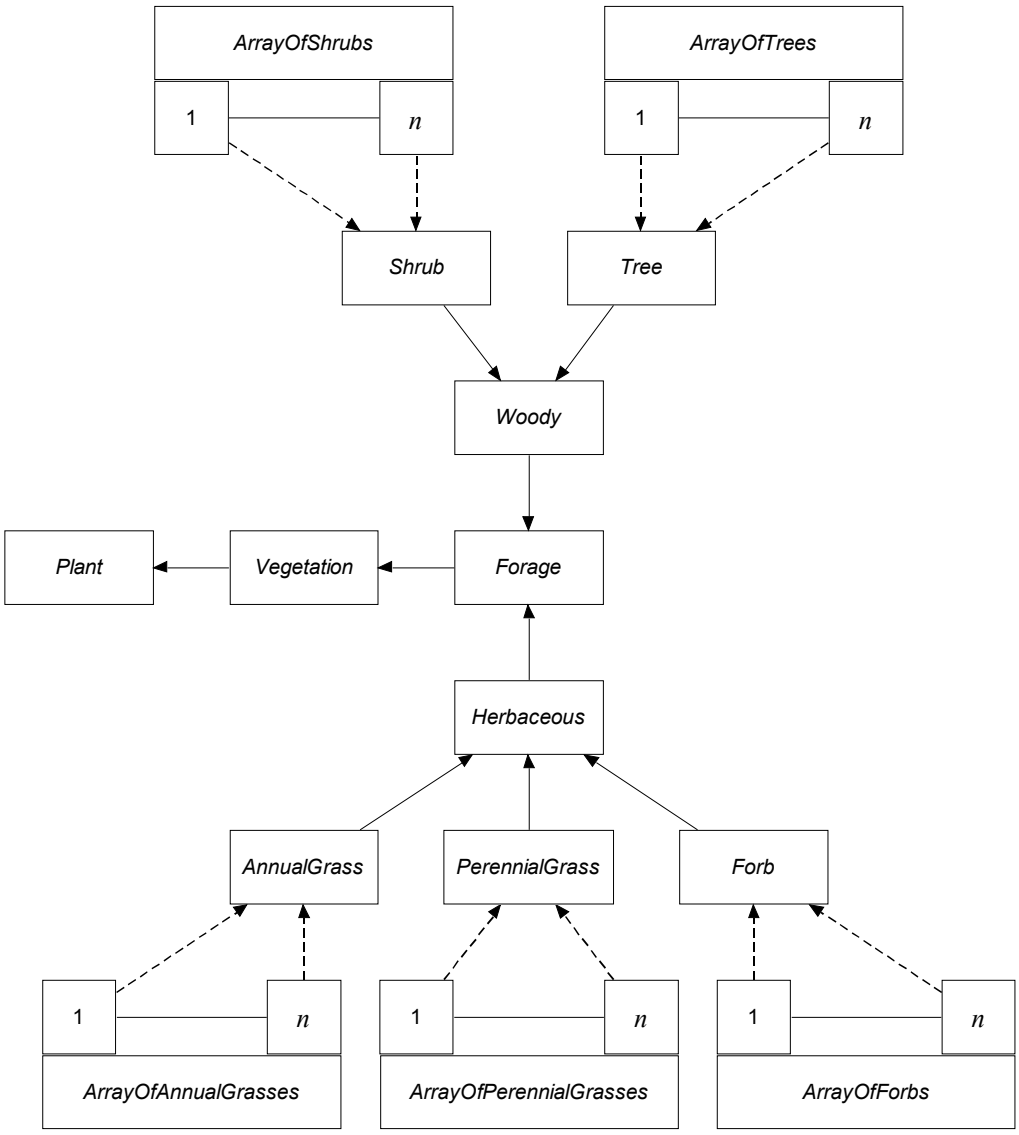
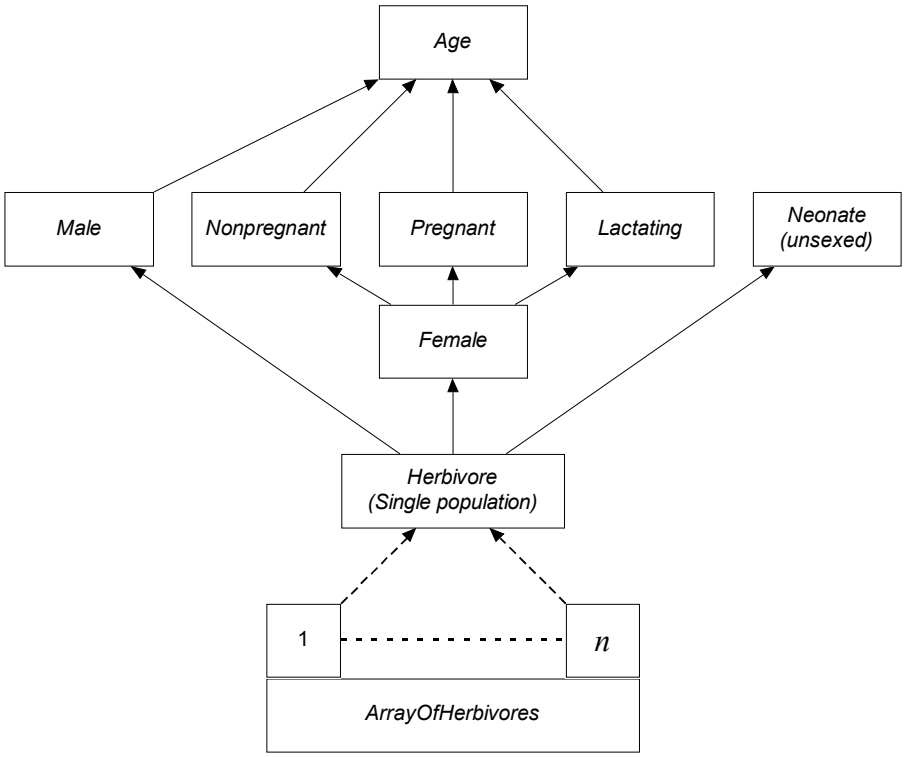

ArrayOfShrubs
ArrayOfTrees
1
n
1
n
Shrub
Tree
Woody
Plant
Vegetation
Forage
Herbaceous
AnnualGrass
PerennialGrass
Forb
1
n
1
n
1
n
ArrayOfAnnualGrasses
ArrayOfPerennialGrasses
ArrayOfForbs

Fig.2

a.


Age
Male
Nonpregnant
Pregnant
Lactating
Neonate
(unsexed)
Female
Herbivore
(Single population)
1
n
ArrayOfHerbivores

b.

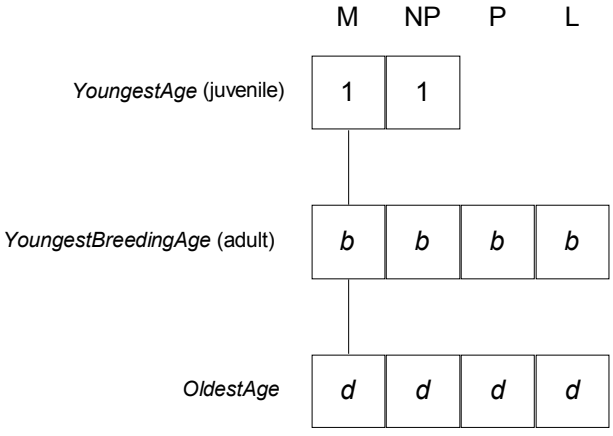

M
NP
P
L
YoungestAge (juvenile)
1
1
YoungestBreedingAge (adult)
b
b
b
b
OldestAge
d
d
d
d

Fig.3

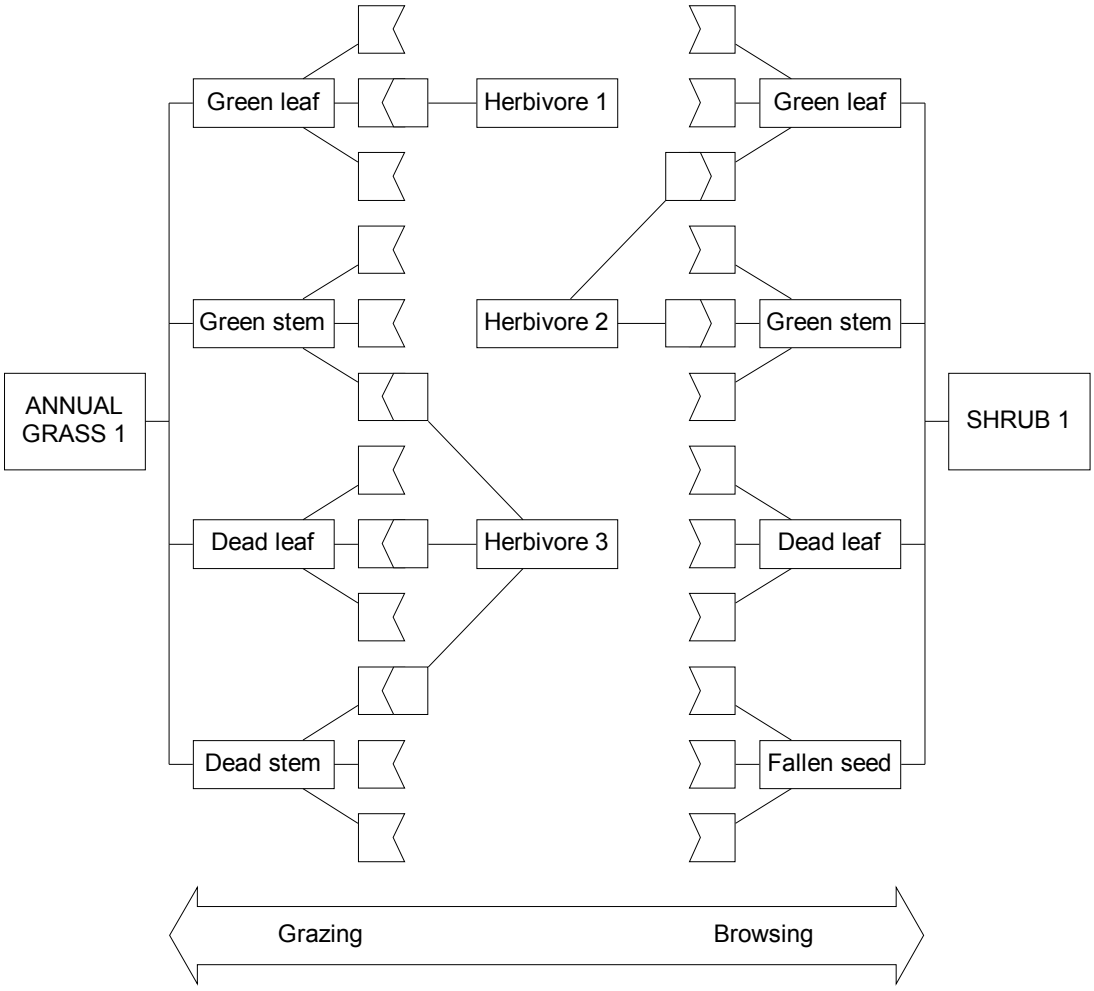
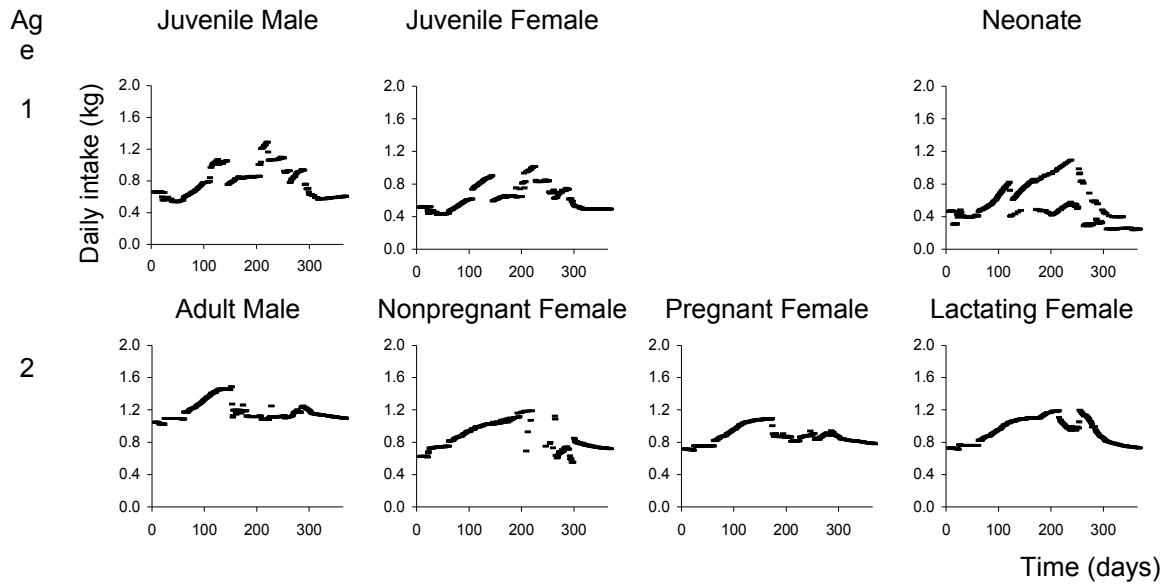

Green leaf
Herbivore 1
Green leaf
Green stem
Herbivore 2
Green stem
ANNUAL GRASS 1
SHRUB 1
Dead leaf
Herbivore 3
Dead leaf
Dead stem
Fallen seed
Grazing
Browsing

Fig.4.

a. Goats



b. Cattle

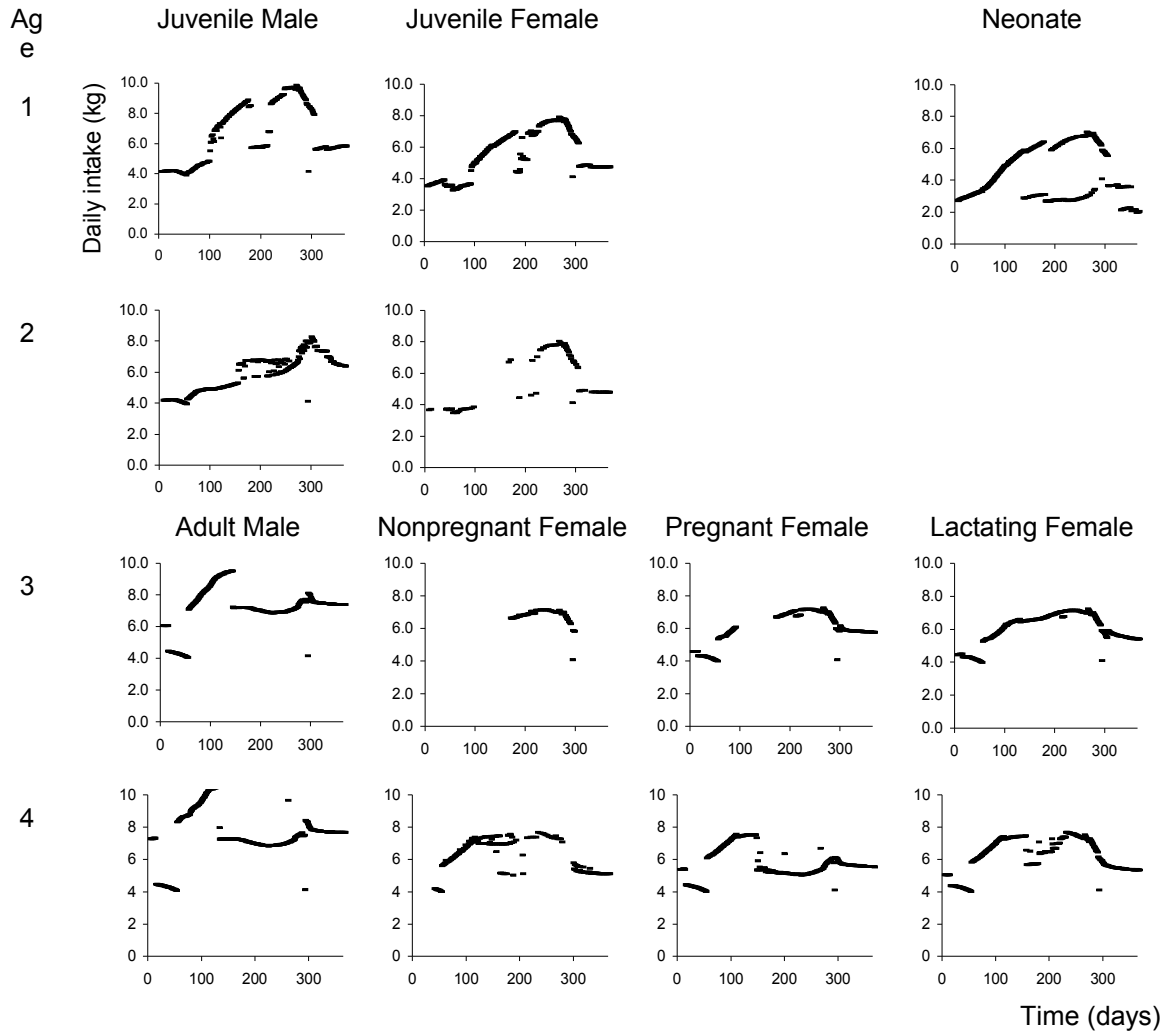


Fig.5.

c.

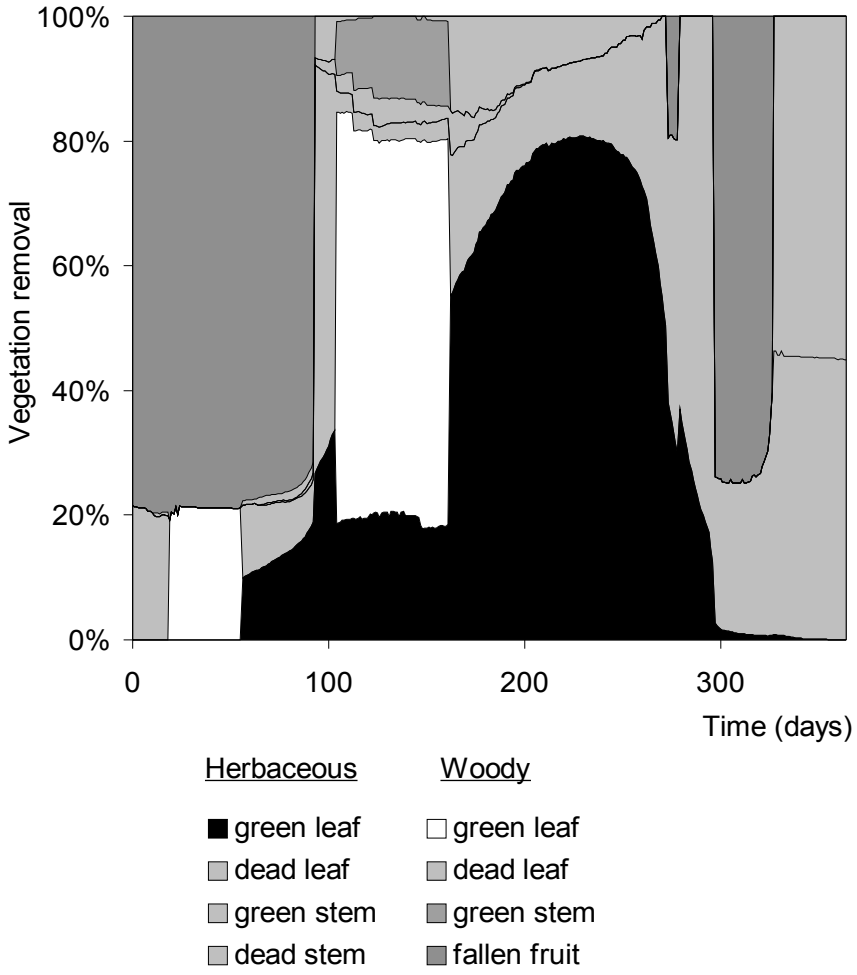

100%
80%
60%
40%
20%
0%
Vegetation removal
0
100
200
300
Time (days)
Herbaceous
Woody
green leaf
dead leaf
green stem
dead stem
green leaf
dead leaf
green stem
fallen fruit

Fig.5.