An Object-Oriented Simulation Model of a Complete Pastoral Dairy Farm

R A Sherlock, K P Bright and P G Neil
Dairying Research Corporation
Private Bag 3123
Hamilton, New Zealand

Abstract The design of an object-oriented framework for the flexible development of a computer simulation of a complete pastoral dairy farm for research use is outlined. The prime aims of the model are to provide maximum flexibility in the specification of farm configurations and management scenarios, and to facilitate utilisation of existing subsystem models, typically written in a variety of languages and software environments, as extensively as possible. Reasons for using an object-oriented approach, and for the choice of VisualWorks Smalltalk in particular, are discussed. The problem of handling the unavoidable combination of continuous-time system properties and discrete events aspects of the real-world system within an object-oriented design, and the solution implemented in this work, is described. The results of simulating a very simple pasture harvesting scenario are presented to illustrate features of the approach.

1. INTRODUCTION

A pastoral dairy farm is a complex system with many poorly-defined relationships between components, particularly the animal-pasture interaction during grazing. The situation is further complicated by major dependencies on uncontrollable variables such as climate which make unambiguous field experiments virtually impossible to perform. As a consequence, there is a clear role for a computational model with a sufficiently detailed level of representation to allow simulation experiments that provide results unbiased by uncontrolled variables. This would minimise the need for, and focus the direction of, costly field investigations of differing stock management systems, supplementary feeding regimes, etc. The process of developing such a model should also help identify and direct research towards critical areas of the farm system which are poorly understood, and over a period of time the model should evolve to become an effective repository of the current level of understanding of the pastoral dairy farm system in a practically useable form.

Little progress has been made to date in modelling the *whole* dairy production process in a *pastoral farm* environment (i.e. where the major feed input is grazed pasture). Although a valuable tool for advisers and farmers is provided in the UDDER model of Larcombe [1989, 1994], this does not aim to implement the level of detail or flexibility needed for many research applications. Detailed and well-verified subsystem models do however exist for some components of the dairy production system, in some cases representing many years of cumulative development (e.g. the MOLLY bovine metabolism model of Baldwin [1995]), and clearly any farm system model should make use of such components wherever possible. In this paper we describe a framework designed to allow the ongoing development of such a whole-farm dynamic simulation model for research use. The model framework is explicitly designed to facilitate incorporation of existing (and future) sub-model components, and to readily allow contributions from third parties.

2. DEVELOPMENT ENVIRONMENT

An object-oriented design of a framework to achieve the above goals has been implemented in VisualWorks Smalltalk [ParcPlace-Digitalk, 1995]. This is a general purpose application framework and integrated development environment based on a language which strictly enforces object-oriented (OO) design principles. The rationale for this approach and the background to some of the key design decisions are discussed in this section.

2.1 Rationale for Object Oriented Design

The principal aims of the OO methodology of software development (e.g. see Meyer [1997], Booch [1994]) are to increase reliability and reduce maintenance costs of large complex systems, and to increase re-usability of software components. These goals are achieved primarily through the data-hiding *encapsulation* property of the abstract data type *objects* which extend the concept of a simple data structure by the inclusion of strictly local procedures or *methods* to manipulate that data. An object's methods are invoked by *messages* sent by other objects (and also by itself), and in a strict OO language such as Smalltalk this is the only way in which the object's internal state can be either accessed or changed. This encapsulation property radically constrains the potential for global accessibility and inappropriate data manipulations that lie at the root of most reliability and maintenance problems in complex procedural code.

High level OO design therefore requires the identification of a set of objects, and the specification of the public messaging interface between them, which implements the desired behaviours. In simulations of real-world systems such as a farm, where the real-world components (cows, paddocks etc)

also implicitly encapsulate much of their functionality, it is appropriate and advantageous to make a high degree of correspondence between those components and the major software objects. The same approach can then be applied iteratively to represent more complex objects in terms of interacting sub-component objects, although the close association between real-world entities and software objects may weaken as abstract functionality is identified and implemented in object form. Once the responsibilities and collaborations of the constituent objects have been specified [Wirfs-Brock et al., 1990] the overall model functionality is independent of the details of the internal representations of those objects and their method implementations. This greatly facilitates development by different contributors and the incorporation of component objects from different sources.

Clearly any self-contained executable program (implementing a component sub-model such as cow metabolism, for example) inherently has the same basic encapsulation property as a software object (although there are some important differences). Its internal structure is hidden, and it can be viewed as receiving messages from the user (which typically consist of command-line and/or keyboard entries), and sending messages (containing its output data) to a file object or screen window object. Thus legacy procedural-code components should, in principle, be able to be incorporated in an OO model by providing some form of interface to adapt their 'messages' to the protocol of the host OO environment. The major OO languages do provide just such a capability, and its utilisation in practice is discussed in detail elsewhere in this conference by Neil et al. [1997].

There are other important object properties such as inheritance and message polymorphism which contribute to effective re-use of components and to the clearer, simpler (and hence more easily maintained and extended) code that characterises a good OO implementation. See Booch [1994] and Meyer [1997] for OO fundamentals; Kreutzer [1986] and Fishwick [1995] for application of OO techniques to simulation models in general. Plant and Stone [1991], Gauthier and Néel [1996], Acock and Reddy [1997], Sequeira et al. [1997], and Lemmon and Chuk [1997] discuss some interesting agricultural and biological applications.

2.2 Choice of VisualWorks Smalltalk

Smalltalk [Goldberg and Robson, 1989] is a prototypical OO language with a rich and stable base class structure resulting from over 20 years of development. Versions are available from two major and several minor commercial vendors, and differ mainly in the extensions (in the form of additional classes) provided to implement their powerful integrated development environments, visual GUI design tools, etc. Automatic garbage collection largely relieves the developer of responsibility for memory management, and incremental compilation brings the advantages of an interpreted language during development while imposing a minimal run-time penalty. Both are standard Smalltalk environment features which greatly facilitate rapid prototyping. The penalty of a typical 8Mb image size is becoming less important with current desktop machine specifications.

Of the various Smalltalk implementations VisualWorks (VW) was chosen for this project, at least for the initial development, on the basis that it was the most mature product with extensive support via vendor and third party class library extensions. It is also the subject of several excellent independent texts. e.g. see Lewis [1995], Howard [1995]. VW has powerful cross-platform capabilities, and the strong separations implicit in its Model-View-Controller partitioning of application functionality should facilitate porting to another Smalltalk, or even another OO language, if the need arises.

Other OO development environments were given serious consideration. Those based on C++ were rejected on the grounds of the language complexity and the onerous programmer responsibility for memory management. Borland's Delphi was considered too great a risk because of its proprietary language (only the Object Pascal version was available at that time) and single source. Also, as these more traditional languages require separate compile and link phases, their development environments cannot support the 'in-line' component testing and dynamic debugging that is such a valuable feature of Smalltalk. We note too that the supposed execution speed advantages that their pre-compilation brings are probably not as great in realistic benchmarks as is often claimed. e.g. see Piraino [1996] and Buck [1997].

3. MODEL OVERVIEW

Our basic representation of the farm is a state-variable (sv) description of a continuous-time dynamical system (e.g. Padulo and Arbib [1974], Woodward [1997]). State variables x_i are always associated with some storage mechanism (of matter or energy), and summarise the past history of the system in the sense that their values at some time t_0 (along with knowledge of the system parameters) provide all the information that is needed to calculate future states (and outputs) on the basis of inputs u_k at $t > t_0$. The basic farm model inputs are sunlight energy for photosynthesis and water from rainfall, specified from time-series meteorological data. Differential equations (DE's), provide the fundamental description of the time evolution of the sv's. Fortunately most of the relevant processes can be well approximated by rates of transfer of matter and energy between different 'lumped element' pools, thus ignoring local spatial dependencies and giving the important reduction to ordinary DE's involving only time derivatives:

$$dx_i/dt = f_i(x_1 \ldots x_N\, u_1 \ldots u_M)$$

In this farm model, fortunately, most of the derivative functions $f_i(.)$ have a direct dependence on only a relatively small subset of the sv's (a consequence of the high degree of compartmentalisation implicit in the biological objects). Even so, these DE's are usually non-linear with time-dependent parameters, and there are many of them. The only practical method of solution for anything but trivial cases is numerical integration, and the procedures to implement this

must be implicit, and preferably transparent to the user, in any simulation model.

In addition there are at least two other requirements which must be recognised. Firstly, the farm configuration (partitioning of total land area into paddocks, specification of plant and animal species, assignment of animals to paddocks etc) is not constant. Discrete management events (such as moving a mob of cows from one paddock to another) changes the actual form of the update equations for the sv's of the components involved. Paddock cover sv's, for example, will need to be updated by DE's describing the grazing process when cows are put in that paddock, instead of the DE's describing normal pasture growth. Secondly, we must cater within the same framework for pseudo-discrete events which cannot be described sensibly by DE's (e.g. machine cutting pasture for conserved feed such as hay or silage).

An outline of the major object relationships to satisfy these requirements is presented in Figure 1. The notation is designed to emphasise the "has-a" relationships as these implicitly identify the messaging paths - an object must hold a reference (direct or indirect) to any other object with which it needs to communicate. It is seen that an object may be referenced (or held by) more than one other object, and that two objects may reference each other.

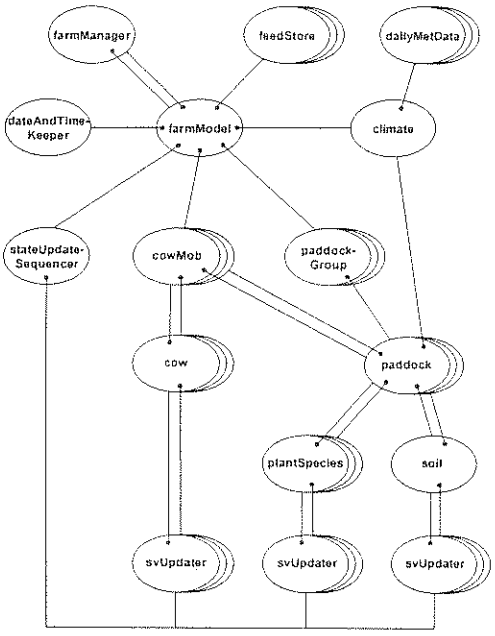

Figure 1: Relationships between the major objects of a very basic farm. Instance variables (denoted by *) hold references to other objects to which it sends messages.

Note that Figure 1 gives no indication of the class hierarchies [Booch, 1994] of the objects shown. However, all except the svUpdaters are instances of immediate subclasses of Model class (which is the base VW class for all domain model objects). Each sv of each dynamical component object has its own svUpdater, the update messaging for these being implemented by stateUpdateSequencer (see Section 3.2). The many different svUpdaters (not distinguished in Figure 1) have considerable common functionality which is implemented in a 3-level abstract class hierarchy rooted in Model.

The cowMob and paddockGroup objects contain collections of animals or paddocks which will be treated as an entity, e.g. dry cows, paddocks available for grazing. Smalltalk's collection classes have powerful methods to iterate procedures over all members of a collection, and to manipulate membership. These give a considerable simplification (and consequent increase in reliability) in the representation of many typical farm operations.

The climate object maintains a collection of dailyMetData records and has methods to read these in from file. Other objects (particularly the paddocks) can thus obtain real historical data or useful aggregations (e.g. 10-year means) on a daily basis.

Other objects implement the mechanics of the simulation procedure, as now described.

3.1 Simulation Time Advance

All simulation clock and calendar information is maintained in the dateAndTimeKeeper object. Having specified begin and end times for the simulation, and the initial states of all the objects, simulation time is advanced by computing sv updates over successive *simulation steps*. These we define as intervals of real time during which the farm configuration (as defined above) remains unchanged. A simulation step will usually be commenced with a configuration change, but this is not a necessary requirement. Specification of the duration of a simulation step, as well as the farm configuration for that step, is the responsibility of the farmManager object (see Section 3.3). Execution of the complete simulation is thereby effected by the following method (see Appendix for brief syntax summary), which also gives some indication of the ability to write self-documenting code in Smalltalk:

```
FarmModel>>runSimulation
  |step|  "Temporary variable"
  [dateAndTimeKeeper atSimulationEnd] whileFalse:
        [farmManager setNextConfiguration.
         step := farmManager nextStepSize.
         stateUpdateSequencer advance: step.
         dateAndTimeKeeper advance: step.
         self updateLoggingData]  "Iterated block"
```

3.2 State Variable Updates

To implement the appropriate update procedures required in different configurational states of the farm, most sv's require more than one svUpdater (although only one is ever active during a specific simulation step). The required svUpdater is activated within the FarmManager>>setNextConfiguration method by adding it to the dependents collection of the

stateUpdateSequencer (and at the same time removing the previously active svUpdater). This makes use of VW's dependency mechanism whereby execution of a 'Model>>broadcast: #aMessage' method causes the same #aMessage symbol to be sent to all of the objects in the receiver's dependents collection. Although this mechanism is peculiar to VW, it could readily be implemented in other Smalltalk dialects (or other OO languages).

Each svUpdater belongs to one of three subclasses, according to whether it implements a singleStep, integration or external update procedure. The first two are implemented entirely in Smalltalk. The external updaters effectively provide an interface to external (usually legacy code) procedures which will provide their own integration routines. Also, this external code will typically maintain many more sv's than need to be accessed by the greater model. These can therefore remain hidden.

The different svUpdater types receive a different sequence of phase messages (from stateUpdateSequencer) which maintain synchronism of the update procedure over the simulationStep time interval. At the start of a new simulationStep each updater receives a #startPhase message. This causes it to make an internal copy of its associated sv (and also copy the duration of the step, which is passed as a message argument). Similarly, the new sv values are copied back by all updaters on receipt of an #endPhase message. The singleStep updaters need only a single intermediate phase message (in response to which they calculate the new sv value). Integrator updaters, in contrast, may require several intermediate phases, the exact number depending on the specified integration algorithm they implement (e.g. 2nd or 4th order Runge-Kutta). External updaters need only initiate their external procedure, awaiting its completion (if necessary) on their #endPhase.

In general the update algorithms will need to reference values of sv's of other objects, and this is done through (read-only) accessor methods of those objects. In this way effective synchronism of updaters is achieved without any constraints on the order in which they are added to their dependency collections (and hence their position in the broadcast sequence). The implementation of this mechanism is further illustrated by the following method code:

```
UpdateSequencer>>advance: stepSize
  singleStepUpdaters broadcast: #startPhase: with: stepSize.
  integratorUpdaters broadcast: #startPhase: with: stepSize.
  externalUpdaters broadcast: #startPhase: with: stepSize.
  singleStepUpdaters broadcast: #phase1.
  integratorUpdaters broadcast: #phase1.
  externalUpdaters broadcast: #phase1.
  integratorUpdaters broadcast: #phase2.
  Integrator type == #RK4 ifTrue:
    [integratorUpdaters broadcast: #phase3.
     integratorUpdaters broadcast: #phase4].
  singleStepUpdaters broadcast: #endPhase.
  integratorUpdaters broadcast: #endPhase.
  externalUpdaters broadcast: #endPhase
```

3.3 Implementation of Management Scenarios

A basic requirement of the model is for a flexible and versatile specification and implementation of different farm management scenarios. This is not a trivial problem, and the degree to which it is satisfied will be a key determinant of the model's practical utility. We have made these tasks the responsibility of the farmManager object, despite the fact that the existence of 'manager' or 'controller' objects in OO designs is often viewed as a warning of over-centralising functionality (instead of distributing it amongst collaborating objects) – Steinman and Yates [1996]. However, we consider such use appropriate in this case as the real-world farm *is* implicitly 'managed' at the top level by a human farm manager who makes decisions on the allocation of feed to animals etc. with the aim of maximising production and utilisation of pasture.

At the beginning of each simulation step the farmManager must therefore specify both the step configuration (principally the assignment of cow mobs to paddocks) and duration, and then send messages to the appropriate objects to achieve this. The approach being taken initially is to provide well-identified and well-named methods in the relevant classes so that the details of what is actually taking place at the farm level are quite clear to a non-programmer such as an agricultural scientist. In effect we are attempting to provide a high-level 'farm management language' in which the setNextConfiguration method can be written by the model user. This should allow a much more flexible means of configuring the farm than use of conventional GUI tools (check boxes, menus, list selections etc). The following code fragment gives some idea of this:

```
FarmManager>>setNextConfiguration
  | milkers nextPaddockInRotation |   "Temporary variables"
  ...
  self moveStock:milkers toGraze:nextPaddockInRotation.
  ...
```

The mob of milking cows and the paddock in which they are to be grazed are first identified from the management strategy (not shown). This will typically be achieved by reference to some pre-determined scheme (e.g. sequential rotation with changes at specific dates and times), application of a set of decision rules or, more likely, a combination of the two. The new configuration is then established by invoking the 'high level' moveStock:toGraze: method. This would be implemented along the following lines:

```
FarmManager>>moveStock:aMob toGraze:aPaddock
  aMob paddock: aPaddock.
  aPaddock cowMob: aMob.
  aMob setGrazeUpdate.
  aPaddock setGrazeUpdate
```

The first two lines link the milkingMob and their new paddock (as indicated in Figure 1). This enables the grazingPaddock to send messages to the milkingMob (and implicitly to all the individual cows in that mob) to obtain the information it needs to update its state over the step interval, and vice versa. The next two lines invoke the (polymorphic)

setGrazeUpdate method of both the cowMob and the paddock. This has the effect of releasing all existing sv updaters in these objects and assigning new ones to implement the update procedures appropriate to a grazing situation. Similar code sequences may set new update associations between other farm components. Note that although the details of moveStock:toGraze: (and its component methods) may be quite complex, that complexity is completely hidden at the level of the setNextConfiguration method.

4. RESULTS and DISCUSSION

The model is still in the early stages of development, so only preliminary results and impressions are available. Nevertheless, these provide support for the general validity of the approach and the desired ability to utilise existing sub-component models has been achieved. This is evidenced by two such legacy components having been incorporated, a pasture growth model of McCall [1989] for which we had access to the FORTRAN source code, and Baldwin's [1995] MOLLY cow model written in ACSL code [Mitchell and Gauthier Associates, 1995]. We note however that modern C++ environments do provide greater capability in this regard – see Neil et al. [1997] for further detail and discussion.

In addition, a simple pasture updater based on a logistic growth equation [Morley, 1968], parameterised from Brougham's [1959] growth data, has been written as a native Smalltalk object. The main purpose of this was to verify the model's internal integration routines, but it gives a sufficiently realistic representation to be useful in more general testing. Selection of one or other pasture sub-model requires a change to only one line in the FarmModel>>build method. A simple illustration of the current stage of development is given in Figure 2.

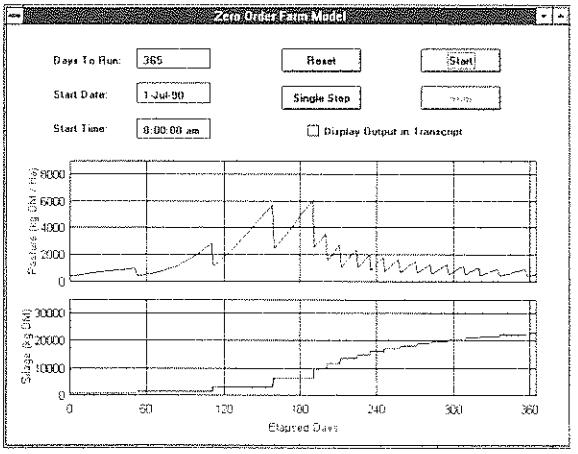

Figure 2: Monitor screen display of pasture cover and silage accumulation over a season.

In this simulation a single paddock is being cut according to a strategy which aims to keep growth rates close to their (time dependant) maxima. The resulting annual production of 23 tonne DM/Ha is well in excess of the best values (15 to 18 tonne) typically achieved in grazing regimes in the same climate. However, the pattern of harvesting necessary to realize this is incompatible with simple stock management.

To date we have not felt any limitation to flexibility through working in a general-purpose OO language rather than a more specialist simulation environment such as DYMOLA [Cellier, 1991], although unquestionably the time spent building the updater structure and integration methods would have been saved by taking that route. However, we excluded that option on the basis of our reservations over committing to such a specialised and proprietary environment, particularly in the absence of any local experience or support.

Although we did not become aware of the work until well into our initial design, and despite the fact that its aim is somewhat different, the basic philosophy and aspects of the implementation of the SAGE system of Gauthier and Néel [1996] appear to have strong parallels with the present work. The possibility of achieving some integration with this framework will be explored.

Performance is not yet an issue, but it will undoubtedly become so as we move from the testing phase to simulations of realistic farms with significant numbers of complex sub-model components. However, we regard that as of secondary importance at this stage of development given the very considerable advantages of the interactive environment that VisualWorks provides. In any case, desktop hardware performance continues to increase and the OO structure and the techniques used to incorporate external components facilitates running the model over multiple machines on a network.

5. CONCLUSIONS

The underlying structure of an integrative framework for a dynamic simulation model of a complete pastoral dairy farm has been described. The ability to incorporate legacy (and future) sub-model components in different programming languages has been a major design criterion. The strongly object-oriented Smalltalk language and the VisualWorks application framework have proved able to meet the design requirements and provided an powerful rapid prototyping and development environment. The OO design principles it implicitly enforces appear natural and helpful in tackling this particular problem, and so far we have not felt any need for a more specific 'simulation language'. The Smalltalk syntax seems particularly suited to providing the basis of an English-like domain-specific language that non-programmers can use to specify complex farm configuration and management scenarios.

6. ACKNOWLEDGEMENTS

This work was supported by the New Zealand Foundation for Research, Science and Technology. We also gratefully acknowledge the major contribution of Leigh Roberts to our understanding of VisualWorks Smalltalk.

REFERENCES

Acock, B. and Reddy, V.R., Designing an object-oriented structure for crop models. *Ecological Modelling*, 94, 33-44, 1997.

Baldwin, R.L., *Modeling Ruminant Digestion and Metabolism*, Chapman and Hall, 578pp., London, 1995.

Booch, G., *Object-Oriented Analysis and Design*, 2nd Edition, Benjamin/Cummings, 589pp., Redwood City, CA, 1994.

Brougham, R.W., The effects of season and weather on the growth rate of a ryegrass and clover pasture. *NZ J. Agric. Res.*2, 283-296, 1959.

Buck, D., Performance benchmarks: Smalltalk vs. C++. http://www.stic.org/BnchMark/D.htm. 1997

Cellier, F.E., *Continuous System Modelling*, Springer-Verlag, 755pp., New York, NY, 1991.

Fishwick, P.A., *Simulation Model Design and Execution – Building Digital Worlds*, Prentice-Hall, 448pp., Englewood Cliffs, NJ, 1995.

Gauthier, L. and Néel, T., SAGE: An object-oriented framework for the construction of farm decision support systems. *Computers and Electronics in Agriculture*, 16, 1-20, 1996.

Goldberg, A. and Robson, D., *Smalltalk-80 – The Language*, Addison-Wesley, 585pp., Reading, MA, 1989.

Howard, T. *The Smalltalk Developers Guide to VisualWorks*. Sigs Books, 624pp., New York, NY, 1995.

Kreutzer, W., *System Simulation Programming Styles and Languages*, Addison-Wesley, 366pp., Sydney, 1986.

Larcombe, M.T., *The effects of manipulating reproduction on the productivity and profitability of dairy herds which graze pasture*, Ph.D. Thesis, University of Melbourne, Australia., 1989.

Larcombe, M.T., *UDDER 8: A desktop dairy farm for extension and research – Operating Manual*. Maffra Herd Improvement Co-op, Victoria, Australia., 1994.

Lemmon, H. and Chuk, N., Object-oriented design of a cotton crop model., *Ecological Modelling*, 94, 45-51, 1997.

Lewis, S., *The Art and Science of Smalltalk*, Prentice-Hall, 212pp., London, 1995.

McCall, D.G., *A Systems Approach to Research Planning for North Island Hill Country*, Ph.D. Thesis, Massey University, New Zealand, 1989.

Meyer, B., *Object-Oriented Software Construction*, Prentice-Hall, 1250pp., Englewood Cliffs, NJ, 1997.

Mitchell and Gauthier Associates, *ACSL Reference Manual*, version 11, MGA Inc, 362pp., Concord, MA, 1995.

Morley, F.H.W., Pasture growth curves and grazing management, *Aust. J. Exptl. Agric. and Animal. Husb.* 8, 40-45, 1968.

Neil, P.G., Bright, K.P. and Sherlock, R.A. Integrating legacy subsystem components into an object-oriented model, in McDonald, A.D., Smith, A.D.M. and McAleer, M. (Eds) *MODSIM97: Proceedings of the International Conference on Modelling and Simulation*, Modelling and Simulation Society of Australia, Hobart, Dec 1997.

Padulo, L. and Arbib, M.A., *System Theory*, W.B. Saunders, 779pp., Philadelphia, PA, 1974.

ParcPlace-Digitalk, *VisualWorks User's Guide*, ParcPlace-Digitalk Inc, 460pp., Sunnyvale, CA, 1995.

Piraino, K., A performance challenge, *Smalltalk Report*, 5(6), 4-11, 1996.

Plant, R.E. and Stone, N.D., *Knowledge-Based Systems in Agriculture*, McGraw-Hill, 364pp., New York, NY,1991

Sequeira, R.A., Olson, R.L, and McKinion, J.M. Implementing generic, object-oriented models in biology, *Ecological Modelling*, 94, 17-31, 1997.

Steinman, J. and Yates, B., Beware the octopus, *Smalltalk Report*, 5(7), 23-24, 1996.

Wirfs-Brock, R., Wilkerson, B. and Wiener, L., *Designing Object-Oriented Software*, Prentice-Hall, 341pp., Englewood Cliffs, NJ, 1990.

Woodward, S.J.R., Dynamical systems models and their application to optimising grazing management, in *Agricultural Systems Modelling and Simulation*, edited by R.M. Peart and R.B. Curry, Marcel Dekker, 1997.

7. APPENDIX – Smalltalk Syntax 101

Statements are executed from left to right, and the basic sequence is always:

```
anObject aMessage.
```

The result of sending aMessage to anObject is always another object, so messages can be cascaded:

```
someNumber asRational printString.
```

- someNumber is expressed as a rational, then printed.

The final returned object can be ignored, as in the two examples above, or assigned to a variable:

```
time := systemClock now
```

- read this as "the variable time gets the object which results from sending the now message to the systemClock object".

Messages can carry one or more arguments:

```
aDictionary at: key put: value.
```

- the method name is at:put: and it requires two arguments.