An Object-Oriented Framework for Farm System Simulation

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

Abstract We describe the structure and implementation of a whole-farm system simulation framework. Developed for building computer models of pastoral farms, the framework design is also applicable to a wider range of farm (and other) systems. The prime objectives are: (i) - to provide the necessary flexibility in the specification of farm components and management scenarios to represent real farm practices, and (ii) - to facilitate utilisation of third-party sub-system component models, typically written in a variety of languages and software environments, as extensively as possible. The basic representation of the farm is a state-variable (sv) description of a lumped-element continuous-time dynamical system. As a consequence of the high degree of compartmentalisation of functionality in the real-world objects, most of the sv derivatives depend on only a relatively small number of sv's of other components – thereby making an object-oriented representation both feasible and appropriate. The framework is implemented in the Smalltalk language which facilitates multi-contributer development through its relative transparency and self-documentation capabilities. Consistent messaging between sub-system component models, possibly running on different networked machines, and the framework is achieved by linking them through proxy Smalltalk objects. The framework imposes a separation of (storable) macro-state objects and component-model-specific updaters. These are combined into the dynamic-component objects which constitute the overall farm model at the time it is instantiated, thereby allowing users (typically agricultural scientists and component-model developers) to make menu selections from alternative sub-component models for state updating.

1. INTRODUCTION

The structure and implementation of a flexible framework for developing simulation models of whole-farm systems is described. Target users are agricultural scientists interested in using computer models to help identify, and then focus research on, system aspects and critical areas of the farm system which are poorly understood. As these are addressed the models should also be capable of evolving to become an effective repository of the current level of understanding of farm systems in a practically useable form.

This rather ambitious goal requires the ability to realistically represent the real-world biophysical entities by component models, to provide flexible representation of their changing configurations and associations, and to provide representations of their interactions. Maximising the ability to utilise component models (both existing and future) developed by third-parties, and maintaining a level of transparency and documentation that makes multi-contributer development a realistic expectation, is essential.

The work has its origins in a simulation model of the New Zealand style pasture-based dairy farm system, Sherlock *et al.* [1997], and that model is currently undergoing validation against farmlet field trials. The underlying Farm System Simulation Framework (FSSF) now described has a wider applicability, however, and that aspect is the focus of this paper.

2. METHODOLOGY

The FSSF is implemented in VisualWorks 3.0 [ObjectShare Inc, 1998]. This is a general purpose application framework and integrated development environment based on Smalltalk – a language which strictly enforces object-oriented (OO) design principles, e.g. Lewis [1995]. The reasons for this choice have been discussed in some detail by Sherlock et al. [1997], but we emphasise the point that the OO analysis and design was developed to increase reliability and facilitate maintenance of large complex software systems, Meyer [1997] and Booch [1994]. This is largely achieved by encapsulating private state and functionality inside 'objects' which can only communicate through a clearly-defined public messaging interface.

The OO approach is particularly appropriate in modelling situations such as the farm systems of interest here in which the real-world entities are inherently quite 'object-like'. For example, an animal has a very complex internal state, potentially needing thousands of variables for an adequate characterisation, but most of these are invisible to

the outside world and the cow's interaction with its environment can be well represented by a small subset of them. Another significant advantage of an OO structure is that external programme modules, typically implementing third-party component models, are inherently object-like and can be 'wrapped' so that they appear as native objects in the framework. Neil *et al.*, [1999] describe this in detail in the context of the FSSF, and Neil [1999] gives further discussion of techniques used to distribute the computational burden over multiple (networked) machines – an important practical issue.

There are several computer languages which support the OO paradigm (with varying degrees of fidelity), and choice of such is always a contentious issue. We have found no reason to change from our original selection of Smalltalk for the framework – if anything its appropriateness has been confirmed. The combination of Smalltalk's large and stable class library, highly automated memory management with garbage collection, immediate expression evaluation and incremental compilation, gives an unmatched development environment. Further, the natural-language-like syntax encourages writing largely self-documenting code – an important consideration for a project with the potential for an extended developmental lifetime and multiple contributers.

Although VisualWorks runs under all major operating systems (including Linux and several Unix platforms) the current FSSF is effectively constrained to Microsoft Windows NT 4.0 because of its extensive use of the Microsoft COM and DCOM protocols [Rogerson, 1997] for linking to external components. The Unified Modelling Language (UML) – e.g. see Fowler and Scott [1997], as realised in the Rose product [Rational Software, 1998], has been used extensively for documentation and analysis, and increasingly as a design tool.

3. FRAMEWORK ARCHITECTURE

3.1 Overview

Figure 1 illustrates the major compositional relationships and associations of a typical farm model implemented in the FSSF. The physical composition of the farm is defined by the components of Land and Animals, which are essentially container classes for multiple instances of animal and paddock objects etc. (This ability to represent individual entities having individual characteristics is essential for modelling 'small' farmlet studies common in the research environment). The Policy class defines management operations while Climate provides the driving inputs (principally for pasture growth) from actual meteorological data. These four major entities (Land, Animals, Policy and Climate) are built from binary file specifications, selected through the upper pane of the user interface shown in Figure 2. This procedure allows 'new' farm scenarios to be readily defined from different combinations of existing entities, and then saved for future recall in an overall 'SimSpec' file.

The lower pane of the user interface provides drop-down lists for the selection of specific component sub-models to be used for updating the states of the dynamic objects in the system, and also for specifying the start date and the time span of the simulation. At present just four subclasses of DynamicComponent have been implemented: Cow, Herbage, Soil and Feedstore - of which sub-model choices are available only for Cow and PastureGrass (a subclass Herbage, which will also have Crop subclasses). However there is no structural limitation in the FSSF to extending both the number of component types and sub-models as the requirement and/or the availability of such increases. e.g. the current Cow is really just a metabolism component – most likely Cow will cease to be a DynamicComponent and become a container of a CowMetabolism, CowUdder and possibly other 'cow-related' DynamicComponents (analagous to Paddock holding Herbage and Soil DynamicComponents).

The FarmManager has the principal responsibility for providing methods which implement 'the mechanics' of altering farm configurations – moving cows between mobs as they change status (e.g. from dry to lactating), assigning mobs to paddocks for grazing, allocating supplementary feed, etc. The scheduling and details of such operations is defined in Policy, which may be a TrackingPolicy which just mimics actual recorded operations on a specific real farm (useful for validation exercises), or a RulePolicy in which algorithmic implementations of a human farm manager's decision rules generate the equivalent information. In both cases this information is contained in a sequence of StepConfig objects, each defining a particular configuration of the farm (composition of groupings, associations of animals with grazing paddocks etc) and the time interval over which this remains unchanged (typically 24 hours or less, depending on how faithfully real farm procedures are being represented). The simulation can then be advanced over the step by attaching appropriate state-update models to each DynamicComponent and executing an update (usually a numerical integration) over the StepConfig interval.

3.2 State Representation

The *state* of a dynamical system summarises its whole past history in the sense that future behaviour is completely defined by knowledge of the current state and subsequent inputs, e.g. Padulo and Arbib [1974]. This concept is central to the FSSF in which the farm

Figure 1. A UML class diagram of the top-level structure of a pastoral dairy farm implemented in the FSSF. Relationships between the shaded elements are expanded in detail in Figure 3. Note that all classes on the same line as DynamicComponent have it as their super-class, and similarly for MacroState – most of the generalisation linkages have been omitted for clarity.
[Notation: The diamond symbol denotes composition. e.g. a CowMob holds (rather than just references) a number of Cow objects. The default navigabiliy of association links is bi-directional, arrow-heads impose restriction (e.g. SubModel objects can send messages to their associated MacroState object, but the reverse is not true). Unless otherwise indicated, association links have a default multiplicity of 1. i.e. the association is between just one instance of each of the linked classes. The symbol * denotes 'more than one'].

is treated as a lumped-element dynamical system (i.e. local spatial dependencies are ignored) by providing a uniform method of representing and updating the states of the bio-physical components which comprise it. Of course, any practical representation of the state of such complex components as cows and pastures can only be a coarse approximation at best. As always, the challenge to the modeller is to find a 'good enough' approximation to adequately represent the range of behaviours of interest for the specific modelling goal.

In the FSSF each DynamicComponent holds a MacroState object comprised of three vectors: IdentityVec, StatusVec and DynamicStateVec (abbrev. DsVec) illustrated in the upper section of Figure 3 for the specific case of a Cow object. The elements of DsVec are the dynamic state variables (sv's). IdentityVec holds constants (e.g. name, breedCode and birthDate for a cow) while StatusVec is typically comprised of flag variables (e.g. calvingDate, isLacting). A MacroState must contain sufficient information to allow its DynamicComponent owner to respond to any requests that other objects in the system may need to make to implement their own functionality. The component vectors of MacroState are instances of Dictionary (a standard Smalltalk class) so that their elements are indexed by symbolic name keys (to aid readability).

As the FSSF is further developed it is likely to require additions to, and/or re-definitions of, the elements of these vectors. It is important that such changes have minimal side-effects in the rest of the system software, and this is ensured by the MacroState being directly visible only to its DynamicComponent owner. Any requests for cow state information (for example) must therefore be handled by methods of the Cow class (or subclass), and if the structure of the CowState vector changes only those Cow method bodies need modifying. Thus although it may be expanded to cater for new state variables, the *existing* public messaging interface is preserved – a benefit of the encapsulation/information-hiding property of OO design.

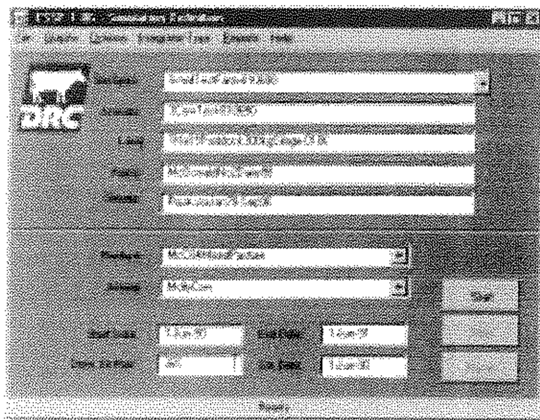


Figure 2. User interface for specification of simulation conditions.

3.3 State Update and Sub-model Selection

To advance a simulation over some time-step it is necessary to update the state vector of each DynamicComponent to a new value based on the existing value and the inputs over that step. In this context inputs may be true external inputs to the whole system, such as climate variables, and/or values derived from the states of other DynamicComponents in the system. The basic description of the time-evolution of dynamic systems is through differential equations (DE's), and in our lumped-element approximation these can always be couched in the form of a set of ordinary DE's involving only time derivatives. i.e. expressing rates of change of sv's in terms of other sv's and external inputs. Although not usually amenable to analytical solution, these can (with some caveats) be readily solved by numerical integration procedures – e.g. Press et al. [1992].

The specification of a particular set of sv's to represent a DynamicComponent, along with the associated update equations, constitutes 'a model' of that object – referred to as a 'component sub-model' in the context of the FSSF. In this work a prime aim has been to utilise existing component sub-models as fully as possible as they frequently incorporate many years of effort in development and validation. The requirement for a flexible assignment of sub-models has a more basic reason too – as the farm simulation progresses quite different updaters will be required for some DynamicComponents. e.g. pasture may be just growing without animals on it, being grazed, or being machine-cut. There is considerable advantage in being able to utilise specific sub-models to represent each of those different situations (instead of hoping to find them all handled inside some monolithic 'universal pasture model').

Figure 1 illustrates the relationship between DynamicComponents, their MacroState and their updater SubModel in the overall FSSF structure. Figure 3 expands this in some detail, illustrating two different updater attachment schemes. All updaters are sub-classed from DsVecUpdater via one of three intermediate abstractions: DsVecExternalUpdater which implements the common functionality of all interfaces to external sub-models, DsVecIntegrator which provides numerical integration schemes (Euler and Runge-Kutta) for native Smalltalk sub-models, and DsVecSingleStepUpdater for cases (like machine cutting of pasture) where a simple 'algebraic update' rather than integration is appropriate. All updaters are dependents of StateUpdateSequencer and so receive a series of sequencing messages controlling copying between MacroState and the local updater state, sequencing numerical integration steps, etc.

The technical issues surrounding incorporation of external (particularly third party) component sub-models into the FSSF has been described in detail by Neil *et al.* [1999] and Neil [1999]; the essence of the solution being the implementation of intermediate layer(s) of software which allow data and commands to be passed between the DsVecExternalUpdater subclass in the FSSF and the environment/application running the component sub-model. Only the Smalltalk object DsVecExternalUpdater is visible from *within* the FSSF, so communication with the external components is by the standard Smalltalk message protocol of the framework.

The other main difficulty in incorporating an external sub-model is the relationship between its own sv's and those of the associated DynamicComponent in the FSSF. There is no simple solution to this, and in some cases the mismatch will be considerable. For example, Baldwin's [1995] very detailed Molly cow nutrition model (which we have successfully utilised) contains over a thousand sv's while the current CowState DsVec contains only ten. However, all ten do have fairly direct equivalents in Molly, so copying these back to CowState (with simple scalings etc. where appropriate) is straightforward. What is less satisfactory is the loss of information on instantiation of a Molly from a specified CowState. Most of Molly's sv's will have to retain their internal default initialisation, and some settling transients must be expected.

Note that the different updater models are always associated with their own concrete subclass of DynamicComponent. This gives them the ability to over-ride default access methods, and hence 'insulate' the rest of the system from variations in usage of different subsets of sv's. As an illustration, PastureGrassState DsVec has seven sv's describing the per Ha dry mass of each of seven physiologically (and nutritionally) distinct components used by the most complex of our currently available ryegrass growth models. We also have access to a simpler

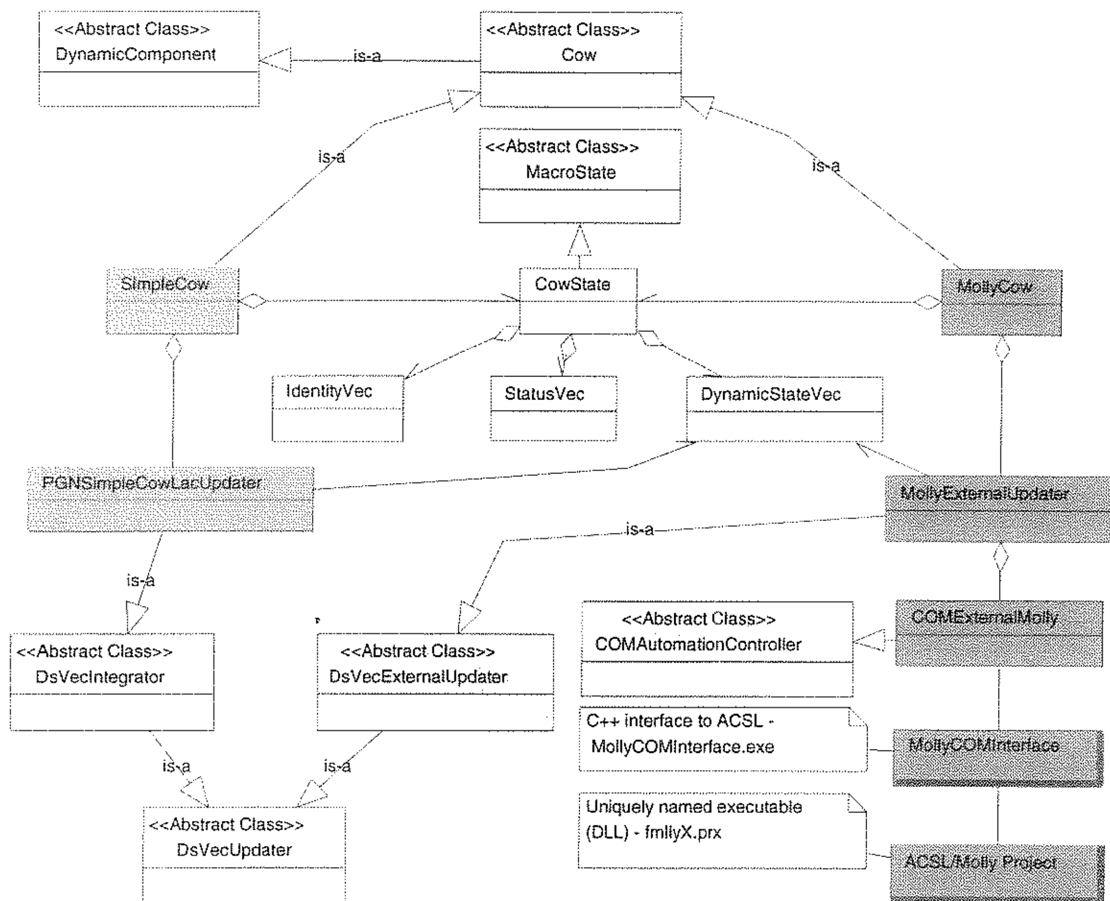


Figure 3. Class diagram illustrating the structure whereby alternative component models can update the state of a DynamicComponent (Cow). PGNSimpleCowLacUpdater is a sub-model implemented in the native Smalltalk environment; MollyCow is a complex external model running in an ACSL environment. Only one of these (along with their associated components) will be instantiated (for each Cow instance) in a particular simulation.

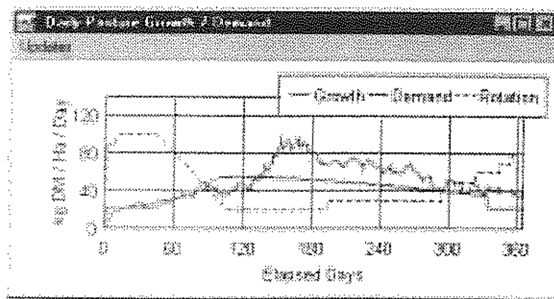
model which makes use of only two of these, and hence the totalAvailableFeedEnergy method in PastureGrass (for example) must be implemented differently in the PastureGrass subclasses implementing the two different growth models. A related issue (not pursued here) is the need to be able to 'normalise' a DsVec to a specific updater model.

3.4 State Storage and Component Instantiation

From the discussion of state in Section 3.2 it is clear that saving (in permanent storage) of just the MacroState of all the instances of DynamicComponent in a specific farm simulation saves enough information to enable the exact same simulation and overall state to be re-created. Since the MacroState objects are little more than data structures, this usefully avoids the need for a specialised object database. State saves can be initiated from the user interface at any stage the simulation is halted or paused, automatically generating the SimSpec files referred to in Section 3.1. Similarly, new farm component specifications can be generated via a user interface which allows convenient form-entry of their MacroState variables and then saves them to file. Instantiation of an overall farm structure from such saved state, with the component sub-model updater classes specified through the user interface at build-time, is straightforward.

4. DISCUSSION AND CONCLUSION

We have presented an outline of a software framework which allows considerable flexibility in implementing a state-based representation of the components of a farm system, and in which different component models, including 'external' third-party sub-models, can be incorporated in a consistent manner. The central issues of state representation and update have been discussed in some detail, but other important considerations remain to be treated elsewhere. These include representation of spatial inhomogeneity within components (potentially



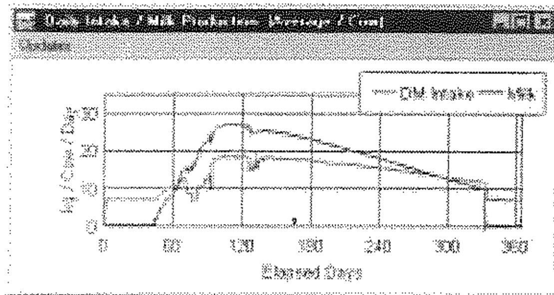


Figure 4. Simulation results showing daily pasture growth & demand, and feed intake & milk production over a season.

important in some pasture growth and grazing scenarios) and improving the ability to represent strongly-coupled components such as paddocks under continuous grazing. The issue of state normalisation for different update schemes also requires consideration.

In the stage of development described here, the FSSF is providing qualitatively realistic simulations of pastoral rotational grazing dairy farming systems as widely employed in New Zealand. Figure 4 shows representative results illustrating how a particular policy for management of the calving distribution and grazing allowance has attempted to match feed demand to pasture production, and the effects of feed shortage on intake and milk production in the early season. A detailed validation exercise comparing the performance of ten experimental farmlets having widely varied management regimes with model predictions is currently in progress.

The FSSF software (source code and all documentation) is publicly available under the terms of the Free Software Foundation's General Public License (the 'GNU Copyleft Agreement'). It is hoped that this will encourage wider participation in both the development of the framework itself and a wider range of component sub-models.

5. ACKNOWLEDGEMENTS

This work has been supported by the New Zealand Foundation for Research, Science and Technology under contracts DRC604 and DRC803.

REFERENCES

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.

Fowler, M. and Scott, K. *UML Distilled – Applying the Standard Object Modelling Language*. Addison-Wesley, 179pp, Reading, MA, 1997.

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

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

Neil, P.G., Distributed Simulation Using DCOM. *Distributed Computing*, 15-18, May 1999.

Neil, P.G., Sherlock, R.A. and Bright, K.P. Integration of Legacy Subsystem Components into an Object-Oriented Simulation Model of a Complete Pastoral Dairy Farm. *Environmental Modelling and Software*, in press. 1999.

ObjectShare Inc, 16811 Hale Ave, Irvine, CA92606, http://www.objectshare.com/. 1998.

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

Press, W.H., Teukolsky, S.A., Vettering, W.T. and Flannery, B.P. *Numerical Recipes in C – the Art of Scientific Computing*, 2nd Edition, Cambridge University Press, 994pp, Cambridge, UK, 1992.

Rational Software Corporation, 18880 Homestead Road, Cupertino, CA 95014, .http://www.rational.com/, 1998.

Rogerson, D. *Inside COM – Microsoft's Component Object Model*, Microsoft Press, 376pp, Redwood, WA, 1997.

Sherlock, R.A., Bright, K.P., and Neil, P.G. An Object-Oriented Simulation Model of a Complete Pastoral Dairy Farm, 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.