A Software Protocol for Distributed Simulation Models

S. L. Wright, V. E. Veraart and B. A. Keating
APSRU,CSIRO Tropical Agriculture, Qld 4350, Australia

Abstract It is much more effective for a scientist to take advantage of an existing simulation system for performing their work since a large amount of the infrastructure, existing models and model components have already been written. However, a major problem with using an existing system is learning how it works, adding your own code to it and interfacing to the existing components. We are overcoming these, and other, problems by defining and implementing a protocol that will enable a scientist to easily add components to any simulation system that complies with the protocol. A simulation system that uses the protocol is structured as a set of components, each handling a well-defined aspect of the simulation, i.e. a set of state variables. A component could therefore represent a spatial or temporal variation in some state variables, or the spatial or temporal dimensions could be distributed over a number of components as appropriate to the system under investigation. An interface description defines the component's state variables and the variables it requires from other components. The code needed to pass data between the components is automatically generated from the interface descriptions and is attached to each component when the simulation is built. A Protocol Manager component is responsible for ensuring that the required variable values are given to the appropriate components at the correct time. Simulation systems that use the protocol can be composed into larger systems in a hierarchical way and simulation components can exist on one computer or be distributed over a network. The requirements for the protocol have been specified and a prototype implementation is being developed and tested by linking two different simulation systems (not currently using the protocol) into a composite system in order to demonstrate the capabilities of the protocol.

1. INTRODUCTION

Simulation is a very powerful tool for modelling and otherwise attempting to understand the structure and behaviour of systems, both natural and manufactured. Therefore it is not surprising that many scientists have used simulation techniques in their particular domain of interest. However since it is often difficult to use existing simulation software, see the discussion in a later section, many scientists have built their own software. This approach causes its own problems with the need to grapple with representational and programming problems as well as the scientific problems. There is also the problem of the 'waste' of resources when many people re-invent the same 'wheels', i.e. the fairly standard mechanisms used in most software simulation systems.

The rationale for the work reported in this paper is driven by two relatively new phenomena in the domain of scientific simulation:

- Increasingly important imperative for scientists to collaborate
- The scale and complexity of the systems to be simulated

These two phenomena are not unrelated. The complexity of some of the systems that people want to simulate requires a range of expertise (both in the science and the software domains) that can rarely be found in one individual, or even in a single research group. The particular work that motivated the approach described in this paper was the requirement for two independent simulation programs, written by independent research groups, in different programming languages, to interact in the simulation of a composite system. Given the already large investment in both software systems it is neither economic, nor feasible, to re-write or translate either or both systems into a common framework, or programming language. In fact it is critically important to maintain the existing scientific validation of the independent systems by altering them as little as possible. Thus we need 'something' to tie together large chunks of existing software into a new simulation system. Rather than resorting to a one-off solution to this particular problem we decided to develop a software protocol and implementation that will enable easy construction of such systems in the future and allow either existing, or new, simulation codes to be integrated in a convenient manner.

2. CHARACTERISTICS OF SYSTEMS TO BE SIMULATED

An important requirement for this work is that it should be able to construct multi-dimensional simulation systems that can be easily reconfigured, adapted and changed. A major problem with many software simulation systems is that if they are constructed with a particular dimensionality, e.g. 1-spatial and a time dimension, then it is often difficult to change them to be, e.g. two-spatial dimensions+time. It is also often difficult to integrate systems with different dimensionalities. In our case we need to combine APSIM and FarmWi$e.

APSIM, McCown et al. [1996], is a Farming Systems simulator that can simulate realistic Farm Management Senarios of crop rotations, etc. for systems analysis purposes. It has a large collection of crop simulation modules that perform bio-physical simulations of particular crops and pastures under different environmental conditions (weather, irrigation, fertilisation, etc.). It uses a single spatial dimension (depth and height above ground) as representative of a paddock, and uses a 1 day time-step over the simulation period (typically ~100 days for many individual crops, but it is often used to simulate 50-100 year cycles of different crop rotations, etc.). FarmWi$e, Moore, et al. [1991], is a Farm-level simulation that has multiple paddocks with pastures plus animals (sheep and cattle) that is used mainly for economic forecasting and as a Decision Support System.

The integrated system requires the use of the APSIM crop modules and management capabilities with the FarmWi$e animal and economics modules. In the future APSIM will be required to simulate 1 ½ or 2 spatial dimensions and variable time steps.

The concept used to integrate the two systems is to isolate the important state variables and activities into components and describe the simulation in terms of interactions between these components. Both existing systems are basically discrete event simulations, however one component used in APSIM (a Soil/Water balance) can be a continuous system inside the component although it interacts with the rest of the system in the discrete event style. Thus the conceptual model of the new system is to have tightly coupled systems within one component (either continuous or discrete event mode) and loose coupling between components (via discrete events).

The current APSIM is a good example of this structure. Each crop has a very complex internal structure (which is, incidentally, difficult to specify with commercial simulation systems) but has a very straight-forward interaction with other parts of the system.

3. RATIONALE FOR A SOFTWARE PROTOCOL APPROACH

There are many possible approaches to constructing simulation systems:

- use commercial software,
- use somebody else's software
- build your own (optionally using existing simulation libraries)

Use of commercial simulation packages is difficult in many cases, for many and various reasons. Many of these packages are, by their nature, very general and thus there is a large gap between the problem domain of discourse and the implementation domain of discourse. Many use proprietary languages, or require you to write code in pre-specified languages and according to their requirements. They often have graphical interfaces which are good to use for simple systems but it is often difficult to express complex models with them. There are often speed problems, cost problems and support problems. The latter is important as we have to distribute our system to ~100 users nationally and internationally.

Using other people's software is also fraught with difficulty: The learning curve is often steep, particularly with the little documentation that tends to be provided with such systems. How is code interfaced into their system, how do you adapt to their way of working and expressing the system to be simulated? Building your own software has its own problems, especially in resource terms, even if you use existing software libraries for particular parts of the simulation.

In our case we have existing software that has to be (re-)used, thus we defined the options as:

- build a super-structure around both systems
- standardise on one system and interface the other to it
- decompose both systems into convenient 'modules' and put them together in a new infrastructure

It was not difficult to decide on the last approach as the most convenient for the current project, and the one with the best possibilities for future evolution and development. To achieve this we needed to separate the 'science', 'simulation' and 'software' aspects of the software. The bulk of the two existing systems would be left alone, or only very slightly modified for interface purposes, as the 'computational' modules (the 'science' parts). A software infrastructure would provide the mechanisms to facilitate the organisation and communication between these parts and a way of specifying the behaviour of the simulation. We define this infrastructure in terms of a **Protocol** - a specification of how separate components can be organised into systems and how they may communicate with each other. The first activity of the joint development group was to define the set of requirements for the protocol that all could agree upon.

4. PROTOCOL REQUIREMENTS

The conceptual basis for the protocol requirements were developed in various ways, including:

- a two-day meeting of 'experts', i.e. the development teams. Both scientists and software experts discussed the essential needs that the software had to meet to be useful, looked at example systems that might be constructed with the protocol and wrote an initial report

- a questionnaire was developed and sent to a large population sample of modellers, simulation developers and users in the agricultural modelling community
- a detailed investigation of the existing systems was performed
- observation and analysis of the use of these systems - both by experts and novices was undertaken

The information was used to generate a requirements document that was circulated for comments within the development group. After a few revision cycles it was released to a slightly wider audience for further comments. After further revision it was adopted as the working requirements document. At this point some people in the group felt the need for a much wider distribution to get external inputs from a wider range of potentially interested parties. However it was decided to construct an initial prototype that could be used to validate the basic ideas before seeking further input, and as a focus of further requirements elicitation.

5. CONCEPTUAL MODEL

We felt the important requirements were for the protocol to:

- be fairly general (but balanced with the need to not exaggerate the semantic gap to the science)
- use the concept of composition of a system out of discrete components
- use a hierarchy to enable scalability of the system
- provide the potential for relatively fine granularity to enable small-scale detailed simulation
- have support tools to build much of the interfacing software automatically

The requirements were expressed in terms of the following logical conceptual model:

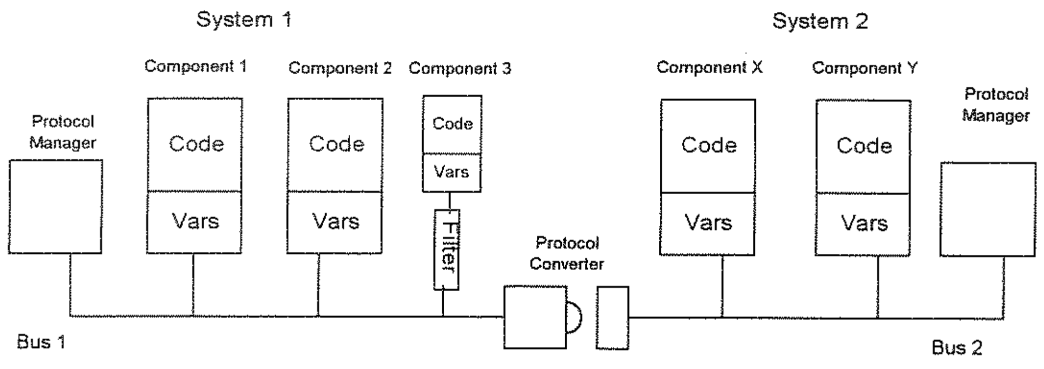

Figure 1

A simulation is composed of a set of Systems (only two are shown in Figure 1) that are connected to a common software 'bus' capable of passing messages and data between the systems. The capability is provided for different Systems to use different protocols (although they obviously must have some basic properties in common in order to interact!) which are translated by a Protocol Converter. In our particular case one system is APSIM and the other is FarmWi$e with no protocol converter as they are both to use the same protocol. Every System is composed of a set of components, also connected to the bus, each handling a well-defined aspect of the simulation, i.e. a set of state variables (represented by Vars in Figure 1). A component could therefore represent a spatial or temporal variation in some state variables, or the spatial or temporal dimensions could be distributed over a number of components as appropriate to the system under investigation. Thus these are the 'computational' or 'science' components of a simulation.

Each component has an interface description which defines the component's state variables and the variables it requires from other components. It also defines the events that a component responds to and generates. The code needed to pass data and events between components is automatically generated from the interface descriptions and is attached to each component when the simulation is built. Each System and each complete Simulation is also defined by an interface description allowing other aspects of the software system to be generated automatically. A Protocol Manager component is responsible for ensuring that the required variable values and events are given to the appropriate components at the correct time. This provides the 'simulation engine' or sequencing part of the simulation. It also handles global tasks such as checkpointing and restarting the whole simulation when required. Each component within a System can itself be a System. This provides the hierarchical composition property that enables many levels of abstraction to be modelled in a convenient manner. Components can exist on one computer or be distributed over a network, allowing for concurrent, parallel computation. Filters can be introduced into the system to perform tasks such as data transformations, e.g. changing units from imperial to metric, accumulation or buffering, etc.

6. ARCHITECTURE

The logical Architecture of the Protocol is shown in Figure 2.

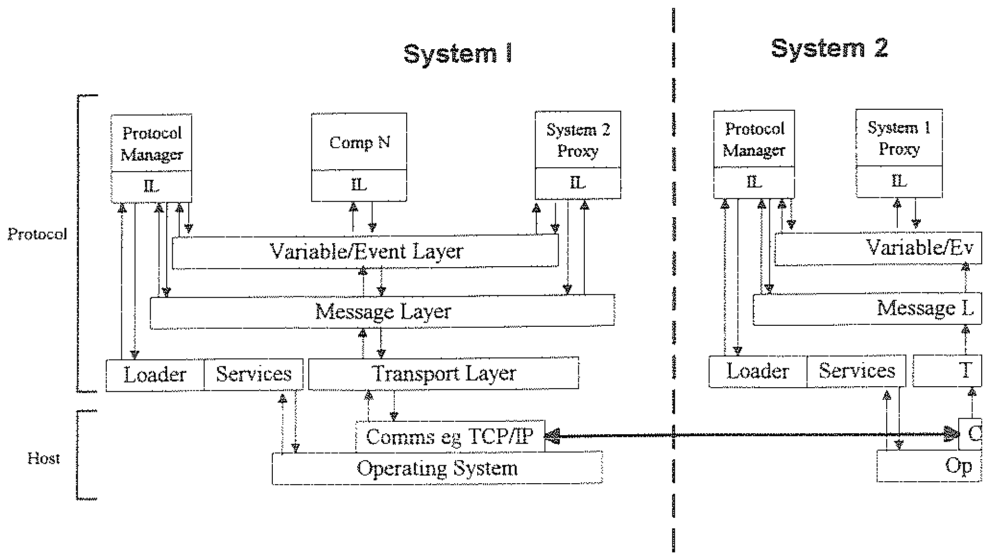

Figure 2

Figure 2 shows two Systems of a simulation executing on two machines on a network. At the bottom of the Protocol Architecture is the Transport and Services Layer - these provide a thin interface to the services (e.g. File System) of the Host Operating System, and its communication services, that the Protocol requires.

The Message Layer provides a message service capable of defining and transporting messages and data structures between components of a simulation. The Variable/Event layer provides a clean abstraction of variables and events to components. Only one component, labelled Comp N, is shown in the Figure but there can be an arbitrary number of them. Each component has an Interface Layer (shown as IL on the Figure) whose main task is to react to each event when it comes into the component from the rest of the simulation. This is basically a look-up table from event to event handler (implemented as a function):

Event 1 param1, param2,... → Comp. fn 1 (param1,...)
Event 2 param1, param2,... → Comp. fn 2 (param1,...)

The IL also handles requests for variable data by translating an external variable name (e.g. System1.CompN.SoilProfile), locating the required internal data and passing this back to the system. The IL for each component is generated from the Component Interface Description in which the component developer has specified the relationship of events to internal functions that handle the event, and the relationships between internal and external variable names.

The Protocol Manager controls the sequence of events within the simulation. The normal configuration is to have event tables controlling the distribution of events. Thus when an event occurs (i.e. is generated in one component) it is sent to the PM. The PM looks up the event tables and distributes appropriate events to other components of the system. Event tables are initially generated at system construction time from the Interface Descriptions of all the components and other system configuration information, however the event tables are dynamic and can be changed while the system is executing. Events and Variable requests for Systems/Components that are not local to a machine are handled by a Proxy Component which can perform optimisations such as caching, etc. Messages between components on one machine does not have to go through the Operating System communications system but are handled by more efficient mechanisms in the transport layer.

7. APPROACH TO THE SOFTWARE DEVELOPMENT PROCESS

This project involves software professionals and is being conducted as a software development project. We have adopted an iterative, incremental and evolutionary/prototyping Software Development Life-Cycle that could be summarised as:

- Relatively complete Requirements and Specification
  - Architecture to meet major Requirements
    - Prototype Design of critical Architectural features
    - Prototype Implementation
    - Iterate
    - Add more design components, prototype, iterate
    - Review
  - Revise architecture if necessary and repeat
  - Review
  - Add further architectural features and repeat
- Revise requirements/specification as necessary
  - Repeat and review
- Add minor requirements, etc.
- Repeat and review, etc.

Experience with this type of approach, Gilb [1988], suggests that such a Life-Cycle will be successful if you have at least 70% of the key requirements in the first iteration (we probably have 90% in this case), along with the major architectural features. We are mainly using an

Object-Oriented approach in our work, but note that this has no impact on the format or structure of the existing code much of which is written in Fortran 77. The need to be able to use the existing code untouched was a primary requirement. The initial development is being done on a PC-based platform with Windows 95 and NT Operating Systems using C++ and Borland Delphi. However the basic communications layer of the prototype interfaces to TCP/IP so there is no reason why versions of the protocol should not be implemented and made available on other popular operating system platforms such as Unix.

8. CONVERTING EXISTING SYSTEMS TO USE THE PROTOCOL

One main advantage of this approach to simulation systems is the ability to re-use existing simulation code. Given that a software simulation code exists, and there is a desire to take advantage of the Protocol facilities, how easy is it to convert the existing code to using the protocol? Naturally this depends on the structure of the existing system. If it is well-structured (in the structured programming or object-oriented sense) then it should not be too difficult to re-organise it into a form suitable for use with the protocol.

The first step is to identify the state variables of the simulation and group them in a way that is logical in the science context of the simulation. These groups become the components in the protocol version of the system. The second step is to identify the key events within the system and find (or create if they do not already exist) the subroutine, procedure or function call that executes the logic pertaining to that event. An interface description can then be written to define the properties of each component and the Interface generator executed to build the interface code. The parts of each component can then be linked into a library, or executable, unit.

The third step is to define the interactions between the components, in terms of the state variables and events, and define a Protocol Manager to coordinate execution of the simulation. This, in general, should not be a difficult task as most simulations usually have some sort of driver, or engine, logic, e.g. the engine code of the existing APSIM software which generates the time events that trigger the daily time-step actions for the other APSIM modules.

As an example let us look at the process of moving the current version of APSIM to use the protocol. APSIM is a single very large executable program but is actually well structured in terms of isolation of state variables. Each crop is in its own conceptual module as are the soil and environmental processes. This structure was designed as a improvement from the ancestors of APSIM (which mixed crop, soil and environment computation) enabling new crops, environmental processes etc. to be easily added to the system. Thus step one has already been attained (Figure 3).

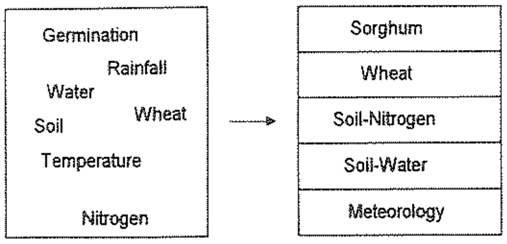

Figure 3

APSIM also has a primitive concept of events but currently these are fixed. At the start of execution of a simulation there is an Initialisation event allowing all the modules to initialise themselves. On each time step there are Preprocess, Processess, and Post-process events. Individual crops also have other events that they respond to such as Harvest, Kill, etc. The distribution of logic for event handling is currently a complex nested if statement in each module with considerable code in each branch (Figure 4).

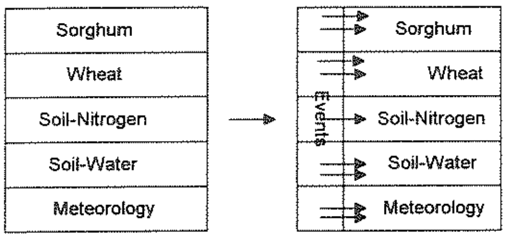

Figure 4

This is easily converted into an event dispatch table by putting all the code for each event into one subroutine and calling it as necessary. Handling of variables is also relatively straight-forward as the common error of sharing variables indiscriminately between modules via FORTRAN common blocks had been avoided. There is a clear protocol of variable exchange between the modules at Pre- and Post- process phases and computation only during the Process Phase. Other simulation codes might require this clean separation of variable access to be performed and debugged in the existing code before converting to adopt the protocol. This process also clarifies the relationships between the modules in the system (Figure 5).

The APSIM modules have now been transformed, by very minor changes in this case, into components, in the sense defined by this protocol. Description files can now be written defining their interfaces to the rest of the system. These descriptions can be used to generate the necessary interface layer logic and we can dispense with the hand-coded APSIM equivalent.

We found that event sequencing (synchronous) was handled by an 'engine' built into the code. In adapting to the protocol this 'engine' logic can be replaced by the event sequencing and distribution of the Protocol Manager.

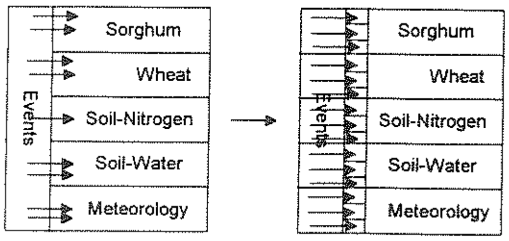

Figure 5

The prototype implementation uses a very simple text file format to describe the interfaces and system configuration information which is similar in structure to the standard .INI file format used by Microsoft Windows, e.g.

Simulation.dsc contains:

```
[Simulation]
name=testsim
system=System1,sys1.dsc
system=System2,sys2.dsc
```

sys1.dsc contains:

```
[System]
name=APSIMv1.5
component=Nwheat,Nwheat.dsc
component=SoilWat,SoilWat2.dsc
component=SoilN,SoilN2.dsc
component=Met,Met2.dsc
...
[VariableMap]
Nwheat.SoilProfile=SoilWat.SoilProfile
Nwheat.NProfile=SoilN..NProfile
SoilWat.Rainfall=Met.Rainfall
...
```

Nwheat.dsc contains:

```
[Component]
name=Nwheat1.9
[In]
var=SoilProfile
var=NProfile
var=temperature
var=SolRadn
event=Harvest
event=Kill
[Out]
var=yield
```

SoilWat2.dsc contains:

```
[Component]
name=SoilWat2.3
[In]
var=Rainfall
var=Temperature
 [Out]
var=SoilProfile
```

Met2.dsc contains:

```
[Component]
name=Met2.2
[Out]
var=Rainfall
var=Temperature
var=SolRadn
```

9. CONCLUSIONS

What differentiates our approach to simulation systems, as described in this paper, to other methods?

- It is designed to take advantage of existing code - not to force re-writing or writing from scratch
- Our system calls the existing software rather than requiring existing software to be adapted to call our routines
- The system works from logical descriptions of the structure and interactions of the components, and tools are used to build much of the software from these user prepared descriptions
- Only the infrastructure is currently provided, adopters can supply their own simulation utilities, or other frameworks, either home built or commercial, to support the simulation aspects

The initial requirements (Wright [1997a]) and architecture (Wright [1997b]) have been well received and we are currently prototyping the various architecture components. We feel that this approach to the construction of simulation systems will prove to be a very powerful one enabling many interesting hybrid systems to be constructed. Many of the problems in doing this work using such an approach are not technical ones to do with software, but interesting scientific issues such as how to interface two systems working on different time-steps, how to handle the different spatial mappings, etc. These we will leave to the scientists to resolve!

10. REFERENCES

Gilb, T., *Principles of Software Engineering Management*, Addison-Wesley, 1988.

McCown, R.L., G. L. Hammer, J.N.G. Hargreaves, D.P. Holzworth and D.M.Freebairn, ASPIM:A Novel Software System for Model Development, Model Testing and Simulation in Agricultural Systems Research, *Agricultural Systems, 50*, 255-271, 1996.

Moore, A.D., J.R. Donnelly, M. Freer, GRAZPLAN:an Australian DSS for enterprises based on grazed pastures. Proceedings International Conference on Decision Support Systems for Resource Management, Texas A&M University, College Station, Texas, USA, April 15-18, 1991.

Wright, S.L., Requirements for MDP Protocol, CSIRO Tropical Agriculture Internal Report, April, 1997.

Wright, S.L., Architecture for MDP Protocol, CSIRO Tropical Agriculture Internal Report, August, 1997.