A Framework Independent Component Design: Keeping it Simple

D. Holzworth[1], N. Huth[1] and P. de Voil[2]

[1] CSIRO Sustainable Ecosystems / APSRU, P.O. Box 102 Toowoomba, Qld, 4350
[2] Department of Primary Industries / APSRU, P.O. Box 102 Toowoomba, Qld 4350
Dean.Holzworth@csiro.au

Keywords: *Model, framework, simulation, component, model interoperability, APSIM.*

EXTENDED ABSTRACT

The bio-physical modelling world is awash with models, modelling frameworks, components and modules. While many were created for specific purposes and have aspects that set them apart from other solutions, there is still considerable overlap between them in terms of functionality, design and data requirements. Many of the models and components aren't compatible with each other and at best, run within one of the major frameworks. This diversity of models and components is desirable though leading to many innovations and new approaches. The downside is much duplication, inefficiencies and wasted time through writing from scratch.

Imagine, for a moment, a world where models and frameworks for a given problem domain are compatible with each other. Modellers could then leverage off many other modellers work much more efficiently and realise the benefits of reuse. Modellers would have more time to focus on science issues rather than writing "yet another wheat model".

There are several options for doing this. Model developers could uniformly adopt one of the major frameworks (there are plenty to choose from) and develop components for that. However this is neither feasible nor desirable. Another option, favoured by the authors, is to develop standard ways of writing a component such that it runs within multiple frameworks and environments.

This paper begins by exploring several modelling frameworks, discussing some sound programming principles that should apply to all model development projects. It then works through one possible design for such a component specification.

Almost as an aside, but much too important to be discarded, is a plea for simplicity. Modellers, model developers, software developers and indeed man-kind tend to gravitate towards complexity. Evidence exists everywhere. This paper reinforces the principle that simple solutions should be favoured over more complex ones wherever possible.

1. INTRODUCTION

There are a plethora of simulation frameworks and models, most of which are incompatible with each other, in terms of science, software and data. For these and other reasons, reuse of models is rarely achieved between frameworks. This then leads to a great deal of wrapping for a target framework but more usually "reinventing the wheel". This rewriting of models is then tightly coupled to a particular framework, further exacerbating the problem.

An option is for modellers the world over to all use a common framework and realise the advantages of model reuse. There are plenty to choose from. The authors are involved in the development of the Agricultural Production Systems Simulator (APSIM), a large farming systems model (Keating et. al. 2003). Others include TIME (Rahman et. al. 2003), OpenMI (Gijsbers et. al. 2005), AusFarm (Moore, 2001) and MODCOM (Hillyer et. al., 2003) to name but a few. However it is neither feasible nor desirable to standardise on a single framework. Diversity of frameworks is always a good thing, providing flexibility and choice to modellers.

The only other way to achieve reuse then is for models to be written in such a way that they run under different frameworks. With some careful design, this is achievable. The modeller then has the advantage of choice. Models and frameworks can be mixed and matched to suite the problem domain.

This paper works through some general software process issues from the agile software foundation and how applying them can help with any software project. It then discusses some of the barriers to developing inter-operable models and explores a simple framework independent component design in detail using some metadata and reflection techniques.

2. AGILE SOFTWARE DEVELOPMENT

When developing any piece of software, for example the component design in this paper, the authors always follow Agile Software principles (http://agilemanifesto.org). These principles describe ways of constructing software to satisfy customer demand. The key principles include continual delivery of software through small iterations, embracing requirements change and building projects around motivated individuals. In addition they promote simplicity – "the art of maximising the amount of work not done".

These principles can be applied to all levels of model development and indeed modelling in general. Model development is a software and science development activity. Agile principles clearly apply to the software development side but can also be useful to science development activities. Simpler science is easier to translate to software and is easier to understand by the model user.

The software team that constructs APSIM, of which the authors are a part of, are continually trying to simplify a very complex software and science framework. This is being done in very small, iterative steps. This process of simplification has itself evolved over time. Many critical tools are now being used (version control, defect tracking etc.) and testing is playing an increasing role in the software process. Huth and Holzworth (2005) describe the many types of tests that have been created and the overall testing process used.

These processes have improved the development of APSIM tremendously over the last several years. Hand-in-hand with this process development is a philosophical shift in thinking among the development team. Simply described as "use common sense", it applies to all facets of feature implementation, defect fixing and software development in general. For example, a large commercial version control system or the development of an extensive suite of tests may not be necessary in very small teams or when the models are small. Time and again, overly complex models and software are presented where a much simpler solution exists. Finding simplicity can be quite difficult given our propensity to gravitate towards complexity.

In terms of the component design outlined below, the authors would favour a collaborate approach between organisations, an iterative process where the design is built up over time and above all else it needs to be kept simple. Indeed all the Agile principles apply to this project.

3. MODEL INTEROPERABILITY

Components are the building blocks of most frameworks. They are typically self contained pieces of functionality that expose an external interface of some kind. The component is then usually compiled into a binary executable (e.g. a dynamic link library .DLL on Windows) that allows dynamic loading by the framework. At run-time, components are instantiated and communicate with each other, either directly or

```
«interface» ILinkableComponent
+ «property» ComponentID() : string
+ «property» ComponentDescription() : string
+ «property» ModelID() : string
+ «property» ModelDescription() : string
+ «property» InputExchangeItemCount() : int
+ «property» OutputExchangeItemCount() : int
+ «property» TimeHorizon() : ITimeSpan
+ «property» EarliestInputTime() : ITimeStamp
+ Initialize(properties :IArgument[]) : void
+ GetInputExchangeItem(inputExchangeItemIndex :int) : IInputExchangeItem
+ GetOutputExchangeItem(outputExchangeItemIndex :int) : IOutputExchangeItem
+ AddLink(link :ILink) : void
+ RemoveLink(linkID :string) : void
+ Validate() : string
+ Prepare() : void
+ GetValues(time :ITime, linkID :string) : IValueSet
+ Finish() : void
+ Dispose() : void
```

Figure 1: The OpenMI IlinkableComponent interface

indirectly through some kind of engine, perhaps via an interface.

This all works well and provides the model developer with the advantages of being able to reuse components from other model developers for that framework. Framework interoperability is not possible though. Components written for one framework are rarely compatible with another framework. APSIM and AusFarm are exceptions to this rule. The first is a cropping systems framework, the second a pasture/grazing framework. The development teams of both frameworks have collaborated on building a binary level protocol that both implement. Components can be interchanged and run along side of each other in either framework.

The advantages of model or component interoperability between frameworks are considerable. Model developers gain access to a much wider range of reusable components and frameworks. Being able to mix and match components and frameworks provides many new possibilities to apply to the problem domain. Modellers could choose many different component configurations, comparing and contrasting approaches.

There are several ways this could be achieved. One approach is that framework builders agree on a single framework and everyone uses that. This is not practical – a single framework is not going to satisfy all problems. A second approach is that a binary 'standard' is adopted in all frameworks. APSIM and AusFarm have adopted this approach. A third approach is that wrappers are built around the foreign component so that they appear to run natively for a given framework. Many frameworks have examples of this. A fourth option, favoured by the authors, is that framework builders choose an implementation platform like .NET and develop a consistent and open component level "standard" that all frameworks use. OpenMI and TIME both specify a component interface using .NET.

Figure 1 shows the interface that all OpenMI components must implement. While fairly straightforward, it isn't completely intuitive for the model developer. For example ID's for components, models and links are not 'natural'. Likewise 'ExchangeItem' is not a problem domain term. What is needed is a more natural way of expressing the functionality of a component.

TIME on the other hand appears much simpler through its extensive use of metadata tags. Figure 2, shows how tags have been inserted into the code to specify inputs, outputs and parameters. There are no references to ID's or linkage mechanisms. The APSIM .NET language binding looks very similar.

```
using System;
using TIME.Core;
public class ToyModel : Model {
   [Input,Minimum(0.0)] double rainfall;
   [Input,Minimum(0.0)] double actualET;
   [State] double netRainfall;
   [Parameter,Minimum(0.0),Maximum(1.0)]
   double coefficient;
   [Output] double runoff;
   public override void runTimeStep( ) {
      netRainfall =
      Math.Max( 0.0, rainfall–actualET );
      runoff = coefficient * netRainfall;
   }
}
```

Figure 2: A TIME example component.

Other paradigms exist that don't rely on reflection and can be just as simple. An application programming interface (API) style of communication could be designed where components directly call a 'get' method to retrieve the value of a variable. This "pull" style of communication is used extensively by APSIM for the non .NET languages where reflection is not possible. Figure 3 shows a portion of the C++ API interface class.

```
class ScienceAPI
  {
  public:
    virtual bool read(const string& name, const
            string& units, float& data, float
            lower, float upper) = 0;
    virtual bool get(const string& name, const
            string& units, float& data, float
            lower, float upper) = 0;
     virtual void set(const string& name, const
            string& units, float& data) = 0;
     virtual void expose(const string& name,
            const string& units, const string&
            description, bool writable, float&
            variable) = 0;
```

Figure 3: An APSIM C++ API for inter-component communication.

The API has methods for reading parameters (equivalent to the [Parameter] in the TIME example), getting the values of values ([Input]) and exposing variables to the simulation ([Output]). These methods can be called by a component at any time. The FORTRAN code in APSIM uses a very similar approach. The cumbersome part of an API like this is the necessity of dealing with different types of data, for example, single, double, integer, strings etc. The APSIM development team choose the simplest approach of just supporting all common data types rather than using more complex c++ templates or variant types. The resultant API is quite large due to the overloading on data type but auto-generation of the API circumvents this to some extent.

Returning to the theme of this paper logically leads to the suggestion of developing a standard for component interoperability. Could a simple standard be specified that deals with the complexities of specifying component inputs and outputs and some other simple entry points like *runTimeStep*?. While such a software standard would go some way towards inter-operable components, inevitably the next issue would be data compatibility.

4. DATA INTEROPERABILITY

Referring back to figure 2 highlights some issues that will need addressing if a framework independent component design is to be built. The *ToyModel* specifies that *rainfall* and *actualET* are inputs, that is, whatever instantiates this component, must supply these two values every timestep. In APSIM, rainfall is called *rain* and means total daily rainfall in millimetres. Is this what the author of *ToyModel* meant when they coded the above example?

What is needed is a way of understanding component data requirements and of matching them with what is available in the target framework. Ontologies are one way of attaching meaning to data. Typically they describe the data, storing the metadata in some kind of database. For large problem domains perhaps a fully populated ontology is necessary, however the authors remain unconvinced of their value for most small to medium modelling projects, preferring simpler methods. In keeping with the theme of simple, iterative style of development, our preference would be to add extra metadata tags to the interface. Data units and a description are probably all that is initially needed. By incorporating this information directly in the source code, close to the implementation, the chances of a mismatch between data meaning and implementation are minimised. Indeed, extracting this information from the source code, via reflection, to create documentation is trivial.

Over time it may be the case that some standardisation does occur on names and meaning of variables in a particular problem domain. For now, simple metadata tagging offers at least some meaning to component inputs and outputs. These could evolve over time, continually being improved. While not a perfect solution, it will at least ease some of the pain of understanding and reusing a foreign model.

Data interoperability can also be thought of as a framework configuration issue. In the *ToyModel* example, *coefficient* has been identified as a parameter. From a component perspective, it doesn’t matter how it gets a value for *coefficient*. It is simply flagging that at some point it will need it. From a modellers perspective though, it matters a lot how these parameters are specified. Modellers typically spend a lot of time creating these parameter sets and reusing them in different simulations. If model interoperability is to be achieved, then some kind of parameter set reuse across frameworks should ideally be supported.

In APSIM, parameters are stored in XML files as in Figure 4. Like text files, this seems the simplest possible mechanism. Of course, elements in the XML file are APSIM specific and will not work in other frameworks even if that framework also reads parameters from XML files. To overcome this issue, either some standardisation of parameter names and XML structure is undertaken for a given problem domain or metadata is added to the XML, similar to the component metadata in figure 2. The authors favour the latter option. Framework interoperability is then achieved by translating the XML for different frameworks. There are many tools and techniques for doing this.

```
<component name="soilwat2"
             executable = “soilwat2.dll">
  <initdata>
    <insoil>2.0</insoil>
    <diffus_const>88</diffus_const>
    <diffus_slope>35.4</diffus_slope>
    <cn2_bare>80</cn2_bare>
    <cn_red>20</cn_red>
    <cn_cov>.8</cn_cov>
    <salb>0.13</salb>
    <cona>2.5</cona>
…
```

Figure 4: A fragment of an APSIM XML simulation file.

Like simulation inputs, data output from a simulation is an important consideration. It would be highly desirable if there was a standardised way of storing simulation output, allowing modellers to use different post-simulation tools. Some frameworks write outputs to databases or spreadsheets. While both these technologies offer some advantages of querying and pivot like summaries, both are vendor dependent. Using the Agile principles of simplicity, text files or XML offer the most transparent and simple storage options. Text files can be viewed and modified using virtually any tool, they can be imported into a spreadsheet and database easily and other tools can very quickly be written to manipulate them. For some situations though, particularly when large spatial datasets are created, text files are not appropriate.

As an aside, the authors have repeatedly seen examples were large databases have been constructed to store observed, field experimental data or other kinds of data. For many field experiments though, a database is overkill. A simple XML file or even a space delimited text file will easily store the several hundred numbers that make up a field experiment. Even a standardised spreadsheet is lower in the complexity scale and more flexible than a database. That’s not to say there isn’t a role for databases. It’s just that simpler solutions should be explored first before adopting a more complex one.

5. TOWARDS A FRAMEWORK INDEPENDENT COMPONENT DESIGN

While the example in Figure 2 is straight forward, simplification is still possible. The example shows inheritance from an interface class called Model. This interface defines the public methods that need to be implemented, for example, *runTimeStep*. While design by interface is a good design principle, indeed Java and .NET make extensive use of it, it may not be necessary, particularly if the interface is very small. Given that extensive use of meta-data tags is already being used, then it follows that the same pattern could be applied for marking the timestep method; e.g. [Timestep].

The distinction between inputs and parameters may also be unnecessary. We believe that a component should simply have inputs. Parameters are a specialised type of input that is read from some type of parameter file. If a component needs a value for *coefficient* then it is the responsibility of the framework to provide that value either from a user interface, a parameter file, another component or someplace else. This provides more flexibility in variable routing.

```
using System;
using Framework.Core;
[Model]
public class ToyModel {
   [Input,Minimum(0.0), Units(“mm”), Description(“Total daily rainfall”)]          double rainfall;
   [Input,Minimum(0.0), Units(“mm”), Description(“Daily evapotranspiration”)]  double actualET;
   [State] double netRainfall;
   [Input,Minimum(0.0),Maximum(1.0), Description(“Rainfall / runoff coefficient”)] double coefficient;
   [Output, Units(“mm”), Description(“Daily runoff”) ] double runoff;
   [TimeStep] public void runTimeStep( ) {
      netRainfall =
      Math.Max( 0.0, rainfall–actualET );
      runoff = coefficient * netRainfall;
   }
}
```

Figure 5: One example of a framework independent component.

Tagging data members as states is only necessary for frameworks that support checkpointing or taking snapshots of a simulation during runtime. Components should be treated as blackboxes and exposing internal data members tends to cut across the principle of encapsulation. Given that checkpointing is a useful function to support, perhaps this tagging needs to remain.

The result of the above refinements provides one possible framework independent design as shown in Figure 5. It's worth noting that much of this already exists in TIME.

The using statement has changed from TIME.Core to a generic term like *Framework.Core*. This assembly simply provides the metadata tags Model, Input, Output, State and TimeStep.

A similar API style of interface where a component calls methods of an interface class could also be designed. Indeed, if multiple languages are to be supported then this may be necessary as well.

For this component design to truly facilitate reuse there needs to be a set of principles developed to support the design. Donatelli and Rizolli (2007) explore some of these.

- Fine grained components are more likely to be reused than larger ones. Care needs to be taken when designing a component that the system boundaries are optimal for a range of frameworks. For example, should a component calculate soil water runoff or should it perform the entire below ground water balance?
- Components depend on their data and so describing the data through appropriate meta-data on inputs and outputs is essential.
- Components should be extensible via inheritance. This needs to be designed into the component to allow this to happen.
- Dependencies should be kept to a minimum. Framework specific API's should be avoided. If utility classes are called, they should be distributed with the component.

As always, an overriding principle applies: *keep it simple!*

6. CONCLUSION

The authors believe there is a need for an alliance of bio-physical model developers to develop a framework independent component design outside of organisational boundaries. Some documentation and principles that accompany the design are also needed. Applying Agile techniques to this project is probably also important. The design should be as simple as possible and evolve over time.

Perhaps a simple, small web site could be created to develop and promote the *standard*. It shouldn't take more than several pages to describe the standard and principles. If it does, then simplicity hasn't been achieved.

The design doesn't need to be as described in this paper. There could well be other ideas that offer more advantages. In some ways it doesn't really matter too much. The important issue is having some agreement across various frameworks on how to design a framework independent component.

This project will only work if there is demand from like-minded model developers. Are there other model development teams that would like to see something like this happen?

7. REFERENCES

Donatelli, M. and Rizolli A., (2007), A design for framework-independent model components of biophysical systems, Proc. Farming Systems Design 2007, Sicily, Sept. 10-12 2007.

Gijsbers, P., Gregersen, J., Westen, S., Dirksen, F., Gavardinas, C., Blind, M., (2005), OpenMI Document Series: Part B Guidelines for the OpenMI (version 1.0), Edited by Isabella Tindall, on web: http://www.OpenMI.org

Hillyer, C., Bolte, J., van Evert, F. and Lamaker, A., (2003), The ModCom modular simulation system, European Journal of Agronomy, 18:333-343.

Huth, N. and Holzworth, D., (2005), Common Sense in Model Testing, Proceedings of MODSIM 2005, International Congress on Modelling and Simulation, Melbourne, Australia, December 2005. Modelling and Simulation Society of Australia and New Zealand Inc.

Keating, B.A., Carberry P.S., Hammer, G.L., Probert, M.E., Robertson, M.J., Holzworth, D., Huth, N.I., Hargreaves, J.N.G., Meinke, H., Hochman, Z., McLean, G., Verburg, K., Snow, V., Dimes, J.P., Silburn, M., Wang, E., Brown, S., Bristow, K.L., Asseng, S., Chapman, S., McCown, R.L., Freebairn, D.M., Smith, C.J., (2003) An overview of APSIM, a model designed for farming systems simulation, European Journal of Agronomy, 18:3-4, 267-288.

Moore, A.D. (2001) FarmWi$e: a flexible decision support tool for grazing systems management. Proc. XIX International Grassland Congress.

Rahman,J.M.,Seaton,S.P.,Perraud,J-M, Hotham,H., Verrelli,D.I.and Coleman,J.R. (2003) It's TIME for a New Environmental Modelling Framework. Proceedings of MODSIM 2004 International Congress on Modelling and Simulation, Townsville, Australia, 14-17 July 2003. Modelling and Simulation Society of Australia and New Zealand Inc., Vol 4, pp 1727-1732