18th World IMACS / MODSIM Congress, Cairns, Australia 13-17 July 2009
http://mssanz.org.au/modsim09

Reflection + XML Simplifies Development of the APSIM Generic PLANT Model

Holzworth, D. [1] **and N. Huth** [1]

[1] *CSIRO Sustainable Ecosystems / Agricultural Production Systems Simulator Joint Venture (APSIM JV)*
Email: Dean.Holzworth@csiro.au

Abstract: The Agricultural Production Systems Simulator (APSIM) is a farming systems model that contains many sub-models. These sub models, in turn, contain many lines of legacy code that often need maintenance and refactoring. One model in APSIM that has undergone considerable reworking is the generic PLANT model, a model capable of simulating many crop species through parameterisation. Originally written in FORTRAN (Robertson *et al. 2002,* Wang *et al.* 2002), this model was converted to the C language and later restructured into classes using C++. This evolutionary process led to a model that was difficult to work with.

More recently, newer computer languages have emerged that have the ability to return, at run-time, various metadata about the source code to a calling piece of code. This reflection, also called introspection, can be used by a model framework to make it more dynamic and able to respond to different types of models at run-time. For example, rather than using inheritance or an interface to locate and call a timestep method of a model, the infrastructure can analyse the model source code, looking for 'tags' that provide information about the appropriate method to call. These tags can also be used to describe the properties and methods of a model, allowing the infrastructure to provide values for them automatically. This can significantly reduce the amount of 'plumbing' code that the model developers must write.

XML is another technology that can simplify the configuration and development of a model. In the APSIM generic PLANT model, it is used to select the desired processes to connect together to define a crop model and then to specify the parameters for those processes. The hierarchical nature of XML lends itself particularly well to this type of model specification. The APSIM PLANT model is reduced to a library of plant classes that describe the various organs and processes required to simulate the growth and development of many crop and tree species. Some of these processes are alternatives to other processes, for example, leaf development can be simulated as a whole of plant process or as cohorts of leaves. The selection of which approach to use for a particular crop is defined in the XML configuration file for that crop.

This paper explores our use of reflection and XML in an attempt to simplify model development. These techniques aren't particularly new or novel in the software development industry, but their use in model development has been limited.

Keywords: *APSIM, APSIM Plant, model development, XML, reflection, documentation*

1. INTRODUCTION

The Agricultural Production Systems Simulator (APSIM) is a farming systems model that contains approximately sixty science (non infrastructure) sub-models (Keating *et al.* 2003). These models simulate a range of crop, pasture and tree species as well as the below ground processes of soil water and nitrogen movement. They are written in a range of computer languages and different programmer formatting styles, due to the eighteen years of APSIM history and the technology changes that have occurred during that time.

APSIM model developers have always had a philosophy of writing models that are decoupled from each other and from user interfaces. This was essential to allow swap in/out capabilities. To enable this swap in/out, a protocol was adopted (Moore *et al.* 2007). While this worked, the science code became littered with calls to the infrastructure to read parameters, retrieve values for input variables and provide values of variables to other models. In recent times, an attempt has been made to move this 'plumbing' code from the model code to the infrastructure so that it is invisible to the model developer (Holzworth *et al.* 2007). We've come to realize that good, clean code is essential for understanding the underlying processes, aiding transparency for others looking to reuse the code.

In recent years, new software development technologies have emerged that allow models to be written with almost no plumbing code at all. Reflection (also called introspection) is a technique that allows a framework to discover, at runtime, metadata about the source code of a model (Holzworth *et al.* 2007; Rahman *et al.* 2004). The Microsoft .NET languages (plus Java and several others) support reflection. This technology allows the model developer to write code very cleanly without the need for calls to an interface to get and set values of variables etc. Another recent development in the software development industry is the eXtensible Markup Language (XML), a text based, hierarchical, file format used extensively in many problem domains. This format can be easily read by source code and can be transformed to other, possibly non-XML, formats allowing new opportunities for using the data in different ways, for example, documentation.

In parallel with these software developments, work on the APSIM generic PLANT model (Robertson *et al.* 2002; Wang *et al.* 2002) has continued, making it capable of simulating around 30 different species of crops from a single code base. The software aspects of this work involved moving the code from FORTRAN to C to the C++ computer language with extensive refactoring to remove plumbing code and improve readability. It has been a continuing goal to convert PLANT to an object-oriented framework, making it simpler to adapt and evolve to changing requirements. On the science side, the source code of PLANT has become a library of process classes that can be turned on and off and parameterised. The parameterisation and class switching has been completely externalized to a configuration document, for example the wheat model in APSIM is simply a configuration document that selects and parameterises a set of classes in the PLANT model. While the end result of this work was an object-oriented generic PLANT model, it became very complex and largely unintelligible to the majority of developers and users. This paper outlines how a new PLANT model was developed from scratch using reflection and XML to produce a simpler, more understandable model. This paper suggests that XML and reflection can be combined to make model development in general much simpler.

2. SMART INFRASTRUCTURE

To support .NET development in APSIM, the infrastructure has been modified to use reflection and XML. It reads a model configuration document, uses reflection to locate the referenced class, creates instances of these classes and passes all parameter values to the created instances. Figure 1 shows the APSIM infrastructure reading the top level XML node of *Plant*. It then locates the class in the PLANT library and creates an instance of it, naming it Vine. For each of the XML elements that it finds nested under this <Plant> node (e.g. Population), it locates the correct field or property in our newly created model instance by looking for the appropriate [Param] tag and pushes the parameter value from the XML file to the *Plant* instance. The process is then repeated recursively for all nested classes specified in the configuration file. For example, when the infrastructure encounters *Leaf* in the configuration file, it will locate a class called *Leaf* and create an instance of it and then recursively look for *Leaf* parameter values.

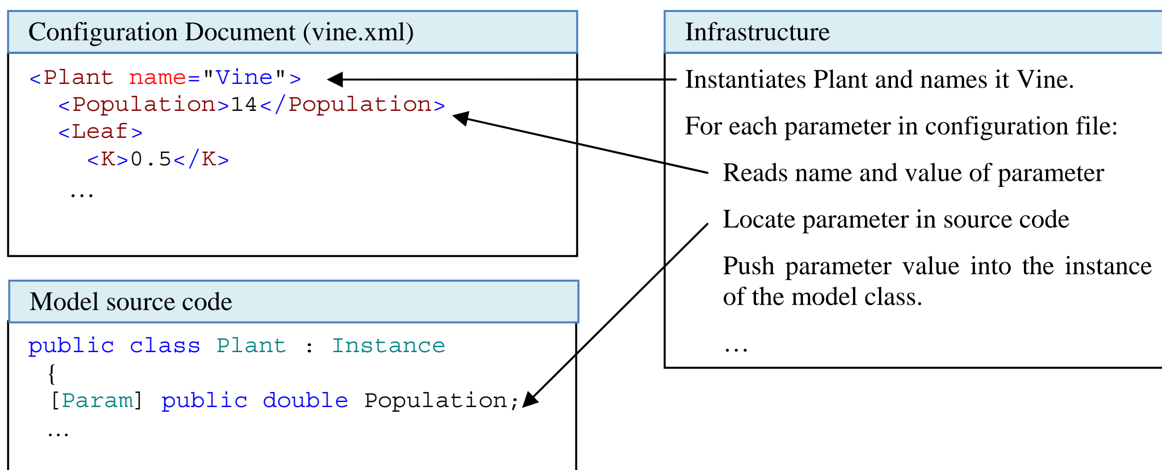


Figure 1: A diagram showing the infrastructure reading the model configuration document and pushing each parameter value into an instance of the model class. Nesting of classes (e.g. Leaf) is also supported.

3. USING THIS CAPABILITY IN GENERIC PLANT

The PLANT model in APSIM uses this technology to dynamically connect processes together at run-time to build a crop model entirely specified from a configuration document. The PLANT model consists of many classes that have been designed in such a way to facilitate their swapping in and out for different crops. As an example, the APSIM *Slurp* model, an *instantiation* of the generic PLANT model, provides a user-defined sink for soil water that can be used to fill the role of a crop within a simulated system, filling the need for a simple surrogate of a crop in a simulation. The *Slurp* model configuration specifies an instance of classes called *SimpleLeaf* and *SimpleRoot*. The *SimpleLeaf* class doesn't dynamically grow leaf; instead leaf area index is supplied as input from the user. Similarly, *SimpleRoot* doesn't dynamically grow roots. Water uptake occurs through a specified extraction coefficient from the whole profile. The other crops in APSIM don't use these classes as part of their configuration. They configure *Leaf* and *Root*, classes that simulate growth in a more dynamic way. The actual instantiation of the classes happens at runtime when the crop model is sown, thus allowing different sowings to instantiate different configurations of the classes.

In addition to configuring whole organs from configuration data, the classes in PLANT have been built to allow processes within an organ to be swapped in and out. Figure 2 shows an excerpt from the vine configuration document, specifically the *ReproductiveOrgan* configuration. The berry filling rate (*FillingRate*) process is configured to be an instance of the *PhaseLookup* class. This class will use the current phenological phase to look up a value. It will iterate through all the nested *PhaseLookupValue* classes finding the instance corresponding to the current phenological phase and returning that instance's *value*. In figure 2, there are two nested classes specified, *EarlyBerryGrowth* and *LateBerryGrowth*. For the former, if the phenological phase is between flowering and veraison, the *value* is itself a function of temperature. When the temperature is between 7°C and 22 ^{o}C, the value is interpolated between 0 and 0.00133. It plateaus at this value until 30 ^{o}C and then drops to zero at 35 ^{o}C. For *LateBerryGrowth*, another temperature function is used but with a different plateau. The key point to note is how easy it is to add and configure these functions. Several simple functions exist in PLANT to return a constant value, perform a lookup on x/y pairs and do interpolations on key variables. These can all be used interchangeably wherever a function is expected. Indeed, in the extreme case, there is a *GenericFunction* that can perform a simple interpolation. X and Y pairs are specified in the XML along with the name of a variable in the model. At runtime, the value of the variable is retrieved and used to perform the interpolation. This provides great flexibility to the model builder as different interpolation schemes can be used without modifying any source code.

New functionality can be added to the model, using these techniques, without any preconception by the model developer. Figure 2 shows how a simple calculation called *WaterContent* was added to the *ReproductiveOrgan* class. A *StageLookup* class configuration was added that specifies the water content for different phenological stages. The sole purpose of this instance of *StageLookup* is to specify a reproductive organ water content that the user, or another model, can retrieve. No source code was added to *ReproductiveOrgan* to support this functionality. *StageLookup* is used in other configurations in different ways reinforcing the case that multiple classes can serve very different roles within the model. These function and lookup classes are very small and simple to write.

```
<ReproductiveOrgan name="Berry">
  <MaximumSize>0.33</MaximumSize>
  <RipeStage>Ripe</RipeStage>
  <GenericFunction name="NumberFunction">...
  <PhaseLookup name="FillingRate">
    <PhaseLookupValue name="EarlyBerryGrowth">
      <Start>Flowering</Start>
      <End>Veraison</End>
      <TemperatureFunction name="Function">
        <XYPairs>
          <XY>7 0.0</XY>
          <XY>22 0.00133</XY>
          <XY>30 0.00133</XY>
          <XY>35 0.0</XY>
        </XYPairs>
      </TemperatureFunction>
    </PhaseLookupValue>
    <PhaseLookupValue name="LateBerryGrowth">
      <Start>Veraison</Start>
      <End>Ripe</End>
      <TemperatureFunction name="Function">
        <XYPairs>
          <XY>7 0.0</XY>
          <XY>22 0.0072</XY>
          <XY>30 0.0072</XY>
          <XY>35 0.0</XY>
        </XYPairs>
      </TemperatureFunction>
    </PhaseLookupValue>
  </PhaseLookup>
  <StageBasedInterpolation name="WaterContent">
    <Stages>StartDormancy EndDormancy BudBurst Flowering Set
    Veraison Ripe LeafFall</Stages>
    <Codes>0.93 0.93 0.93 0.93 0.93 0.85 0.7 0.7</Codes>
  </StageBasedInterpolation>
</ReproductiveOrgan>
```

Figure 2: An excerpt from the vine configuration document showing how berry filling rate is configured as a phenological phase base lookup. Different functions are used for early and late berry growth.

4. WHAT DOES IT MEAN FOR THE MODEL DEVELOPER?

Given sufficient base functionality in the available classes in PLANT, the model developer is able to construct new crop models entirely from configuration documents. This assumes the classes in the PLANT framework are suitable for the new crop model. It is inevitable though, particularly in the early developmental phase that new processes and organs will need to be added to the PLANT framework. This is made much easier through the almost complete absence of any ‘plumbing’ code. Consider the source code excerpt from the *ReproductiveOrgan* class in Figure 3. Unlike earlier APSIM model code written in older languages, there are no calls to ‘get’, ‘set’ or other infrastructure interfaces which greatly simplifies the source code. Inputs and parameters are tagged as such with the infrastructure using them to locate the declaration and supply values at the appropriate time. For example, before the infrastructure calls a method of the model, it will retrieve values of all inputs automatically. The model developer doesn’t know or care where the values come from. Doing this incurs a small runtime overhead but the result is simpler source code, which is our primary goal.

An inheritance interface (*BaseOrgan)* is used by all model classes, but instead of being an infrastructure based interface (calls to read parameters, get and set variable values etc), the base organ class defines how the science in this class interacts with other classes in PLANT (science domain). For example, the base organ defines how the water and nitrogen resource arbitrator communicates with an organ. This frees the developer to think about science based concepts like water uptake, dry matter demand rather than solution space concepts like the mechanics of getting and setting variable values. Our experience has shown that this leads to a model that is much simpler to write and understand.

```
class ReproductiveOrgan : BaseOrgan, Reproductive, AboveGround
   {
   [Event]  public event NullTypeDelegate Harvesting;
   [Input]  private int Day = 0;
   [Input]  private int Year = 0;
   [Output] [Units("/m^2")]  private double Number = 0;
   [Param]  private double MaximumSize = 0;
   [Param]  private string RipeStage = "";
   private bool _ReadyForHarvest = false;
   [Output] [Units("g/m^2")] double LiveFWt...
   public override void DoActualGrowth()...
   public override double DMDemand
      {
      get
         {
         Function FillingRate = Children["FillingRate"] as Function;
         if (Number == 0 && FillingRate.Value > 0)
            {
            // We must be on the first day of filling
            Function NumberFunction = Children["NumberFunction"] as Function;
            Number = NumberFunction.Value;
            }
         if (Number > 0)
            {
            double demand = Number * FillingRate.Value;
            // Ensure filling does not exceed a maximum size
            return Math.Min(demand, (MaximumSize - Live.Wt / Number) * Number);
            }
         else
            return 0;
         }
      }
   public override double DMRetranslocation...
   public override double DMAllocation { set { Live.StructuralWt += value; } }
   [EventHandler] private void OnHarvest()...
   [Output] private int ReadyForHarvest...
   [Output][Units("g")] private double Size...
   [Output] [Units("g")] private double FSize...
   }
```

Figure 3: The source code for *ReproductiveOrgan* shows how nested processes are accessed (e.g. *FillingRate*). The reflection tags Input, Output, Param and Units can also be seen. For more information on the available tags, see Holzworth *et al*. (2007)

Another advantage to the developer arises from using XML as the configuration document format. XML lends itself particularly well to transformation to other forms. As the XML configuration documents are reasonably self describing, they can easily be converted to model documentation which helps the model developer and users alike. APSIM includes a tool that transforms the XML into HTML documentation. Rather than using XSLT language to perform the transformation, a simple custom tool was built so that graphs of the functions could be added to the HTML documentation. Figure 4 shows an excerpt from the *ReproductiveOrgan* documentation that was generated from the XML configuration document. Through some simple rules and the use of a charting tool, the configuration files can be auto-generated into very useful documentation.

Berry

MaximumSize = 0.33

RipeStage = Ripe

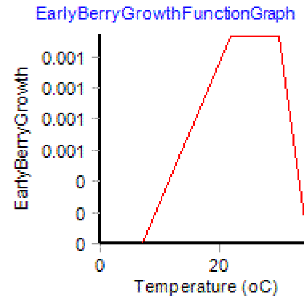
...

FillingRate

EarlyBerryGrowth

The value of EarlyBerryGrowth during the period from Flowering to Veraison is calculated as follows:

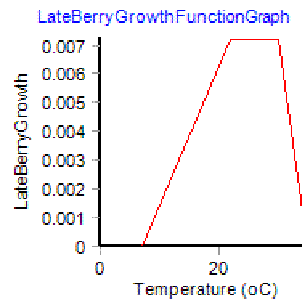
Temperature (oC)	EarlyBerryGrowth
7	0.0
22	0.00133
30	0.00133
35	0.0



LateBerryGrowth

The value of LateBerryGrowth during the period from Veraison to Ripe is calculated as follows:

Temperature (oC)	LateBerryGrowth
7	0.0
22	0.0072
30	0.0072
35	0.0



WaterContent

Stages = StartDormancy EndDormancy BudBurst Flowering Set Veraison Ripe LeafFall

Codes = 0.93 0.93 0.93 0.93 0.93 0.85 0.7 0.7

Figure 4: The auto-generated documentation for the *ReproductiveOrgan* class in Figure 2.

5. DISCUSSION AND CONCLUSION

It is quite possible to use model configuration documents in a model without having to use reflection, as the last 30 years of practice have shown. The model would need to contain hard coded source with large switch or if statements to push the parameter values to the model. It would also need to know about all parameters in all classes. For a model the size of PLANT this quickly becomes cumbersome. Every time the model developer creates a new process or new parameter, the infrastructure would need to change as well. For older computer languages that don't support reflection, this may be the necessity, but for modern languages that have reflection, model development can be made much simpler.

It is also quite possible to have configuration documents in some other format. For example, the older Windows INI format could be used but they suffer from a lack of nesting of sections. This could be overcome by devising a hierarchical scheme for INI files that provides the required nesting. Likewise, a completely custom ASCII format could be developed to configure the PLANT model. In both cases though, devising a custom format, and the code necessary to read and transform it, would be wasted effort when a simple, industry standard exists that meets our requirements.

There are many advantages to the techniques discussed in this paper. Our goal was to simplify the model source code as much as possible, removing all infrastructure plumbing code and user interface specific code. This in turn frees the model developer to focus on the science issues of implementing the model processes. When this source code is then directly configured from configuration files, a much greater level of transparency for model users is achieved. By examining the generated documentation of the model, users can attain a rudimentary understanding of the processes used in a model and how they are configured. This helps to reduce the 'black box' problem. Our experience has shown that this is necessary in order for users to be confident that the model output makes sense and thus increase their understanding of the problem domain. This is, after all, the goal of any model.

The lack of 'plumbing' in the model source code also facilitates model reuse in other applications. The relatively loosely coupled *ReproductiveOrgan* presented in this paper, could easily be incorporated into other models or applications. It is completely decoupled from the APSIM infrastructure, user interface and other APSIM models. It simply specifies its input requirements, the outputs that it makes available, and the methods that need to be called. It is true that the science works in a particular, specific to APSIM way, but so long as this stays relatively simple, this could be mimicked by other models. This science specification describes how PLANT's resource (dry matter, water, nitrogen) arbitrator communicates with the organs. Considerable discussion is currently taking place around the design of these interfaces. This is a key indicator that the infrastructure issues have largely been solved. For an in depth analysis of a real world crop model (broccoli) that uses these techniques, see Huth *et al.* (these proceedings).

A downside to the approach outlined in this paper, is the tendency for the model developer to be overwhelmed by the large number of classes, functions and processes that can be configured. It can be difficult to know where to start when creating a new crop model. Our recommendation is always to start with an existing configuration and modify that but the issue of knowing what classes are available still stands. Good documentation, something we've always struggled to find the time to do well, would help. More auto-generated documentation from the source code might also help. It's something that needs to be addressed in the near future.

Reflection and XML are widely used in the software development industry but their adoption in the field of farming systems modeling has been limited. Indeed, we are not aware of any other work in this area. Reflection and XML configuration documents are not intrinsically tied to APSIM but instead are simple enough to be utilised in all .NET models and model frameworks. The benefits to be gained are considerable while the implementation costs of adopting them are minimal.

6. REFERENCES

Holzworth DP, Huth NI, DeVoil P (2007) A Framework Independent Component Design: Keeping It Simple. *In Oxley, L. and Kulasiri, D. (eds) MODSIM 2007 International Congress on Modelling and Simulation. Modelling and Simulation Society of Australia and New Zealand, December 2007.* http://www.mssanz.org.au/MODSIM07/papers/18_s56/AFrameworkIndependent_s56_Holzworth_.pdf.

Huth, N., Henderson, C. and Peake, A. (2009) Exploring irrigation management of a horticultural crop using APSIM. *In R. Braddock et al. (eds) 18th IMACS World Congress - MODSIM09 International Congress on Modelling and Simulation*, December 2009. ISBN: 978-0-9758400-7-8.

Keating BA, Carberry PS, *et al.* (2003) An overview of APSIM, a model designed for farming systems simulation. *European Journal of Agronomy* **18**, 267-288.

Moore AD, Holzworth DP, Herrmann NI, Huth NI, Robertson MJ (2007) The Common Modelling Protocol: A hierarchical framework for simulation of agricultural and environmental systems. *Agricultural Systems* **95**, 37-48.

Rahman JM, Seaton SP, Cuddy SM (2004) Making frameworks more useable: using model introspection and metadata to develop model processing tools. *Environmental Modelling and Software* **19**, 275-284.

Robertson MJ, Carberry PS, *et al.* (2002) Simulation of growth and development of diverse legume species in APSIM. *Australian Journal of Agricultural Research* **53**, 429-446.

Wang E, Robertson MJ, *et al.* (2002) Development of a generic crop model template in the cropping system model APSIM. *European Journal of Agronomy* **18**, 121-140.