

Making Frameworks More Useable: Using Model Introspection and Metadata to Develop Model Processing Tools

J.M. Rahman^a, S. Seaton^a, S.M. Cuddy^a

^a*Cooperative Research Centre for Catchment Hydrology, CSIRO Land and Water
GPO Box 1666, Canberra, ACT, 2602, Australia, joel.rahman@csiro.au*

Abstract: Several modern development environments allow executable components, such as hydrologic models, to carry Metadata describing the properties and capabilities of the components. These metadata may be restricted to the names of properties, and their respective data types, or may extend to other information, such as classification of properties (eg. input or output), numeric constraints on parameters (eg. between 0 and 1, or greater than 0) or aliases (eg. rainfall, also known as precipitation). Introspection in these environments allows tool developers to write programs and other components that make use of these metadata to provide generic model processing tools, while allowing model developers to take advantage of these tools without additional development effort. Typical model processing tools include model integration systems, parameter optimisers, automatic user interface generation and automated IO. One approach to implementing model introspection and metadata, used by the Interactive Component Modelling System (ICMS), is to extract information from a model when compiling a custom modelling language. An alternate approach, being evaluated in a new framework, relies on the language independent introspection provided by the .NET environment. These uses of introspection streamline model development within modelling frameworks, reducing the effort required to take advantage of other framework capabilities, such as dynamic visualisation.

Keywords: Introspection; Model Metadata; .NET; ICMS; TIME

1 INTRODUCTION

Several development environments, including Java and .NET, include introspection, where compiled components carry metadata, describing their properties and capabilities, that can be discovered, at run-time, by other parts of an application [Meyer 2001][Flanagan 1999]. The included metadata may be limited to the names and data types of properties, which can be derived directly from the source code, or may extend to additional *tags* such as classifying properties as input or output, or numeric constraints on inputs. Programs that investigate these metadata can provide generic functionality that automatically adapts to different components. These metadata can also form the basis of structured reference documentation for a component.

Including a rich set of metadata with environmental modelling components allows the development of a range of *model processing tools*, such as user interface generators, model integration tools and IO schemes for saving and loading model configura-

tions and state. Incorporating introspection capabilities and model processing tools in modelling frameworks, promotes the development and use models within the frameworks.

Modelling Frameworks make model development simpler, by providing support components and structure to handle common tasks, and improve the model user's experience by providing sophisticated visualisation and interaction components. The traditional view of frameworks is as service and structure providers, with developers using a framework by writing components and applications that make use of these capabilities. Introspection reverses this approach, by viewing the variable components, such as models, as providers of services, which the framework is able to access and manipulate.

A common pattern in developing with frameworks is applying a generic framework capability to each property of a component, such as registering each property of the model with a generic framework IO system. Usually this pattern requires little reasoning

from the developer and becomes repetitive. Using introspection to automate this process, by allowing the framework to extract information about components, removes the need for this repetitive code and eliminates the risk of the repetition leading to inconsistencies and bugs. This, in turn, makes it easier for developers to use a framework, and results in modelling components that are more robust in the face of changes to the underlying modelling environment.

The Interactive Component Modelling System (ICMS)[Reed et al. 1999] is an integrated model development environment that allows models to be implemented using a custom modelling language, known as MickL. The MickL compiler extracts metadata from the source code, for use by the ICMS runtime system. ICMS uses this metadata in several tools, including a drag and drop canvas for integrating modelling components.

A new framework, currently known as The Invisible Modelling Environment (TIME) makes use of the extensible, language independent, metadata capabilities of the .NET environment. TIME is intended to be a lightweight framework that imposes very little framework overhead and constraints, making it almost *invisible* to the model developer. TIME is designed to support a wide variety of model processing tools, and currently includes several tools including user interface generators.

Frameworks that use model metadata, to support model processing tools, encourage developers to document models by providing practical improvements to the utility of documented models.

2 ICMS

The Interactive Component Modelling System (ICMS) is an integrated model development environment allowing users to design, implement, test and integrate modelling components [Reed et al. 1999]. ICMS supports model implementation using MickL, a C like, custom modelling language. Components compiled in MickL can be configured, executed and integrated with other components using a graphical, drag and drop canvas. Models are scheduled at runtime using the Open Modelling Engine (OME) [Rizzoli et al. 1998]. Figure 1 shows the implementation of a simple runoff coefficient model in MickL. MickL uses implicit typing, with variable names starting with an uppercase character (such as `Rainfall` or `PET`) representing properties, of the OME class, that are extracted by the compiler for introspection. Because MickL does not use type

declarations, the metadata available to ICMS only includes the names of properties.

```
function Main()
{
  Runoff =
    Coeff *
    max( 0.0, Rainfall - PET );
}
```

Figure 1: Simple runoff coefficient model implemented in MickL within ICMS

3 TIME

The Invisible Modelling Environment (TIME) is a modelling framework designed to simplify model development, while supporting the integration of model components written in different languages. TIME runs on the .NET platform and uses the .NET metadata facilities to extract properties, data types and custom metadata tags from modelling components. Figure 2 briefly describes the custom metadata tags recognised by TIME.

Input Property is an input

Parameter Property is used as a parameter

Output Model property is an output

State Property is an internal state variable

Minimum Property has a minimum allowed value

Maximum Property has a maximum allowed value

WorksWith This component, such as an IO routine or a visualisation tool, works with a particular class of Data

Ignore This property should be ignored by TIME metadata analysis

Figure 2: TIME Model Metadata Tags

TIME is designed in a number of distinct layers, with layers towards the top of the system using services provided by lower layers (Figure 3). With a thin kernel layer, most framework functionality, such as user interface generation and model linking, is implemented in the Tools layer, allowing models to be remain independent of these tools.

Figure 4 shows the runoff coefficient model, from Figure 1, implemented as a TIME model in C#.

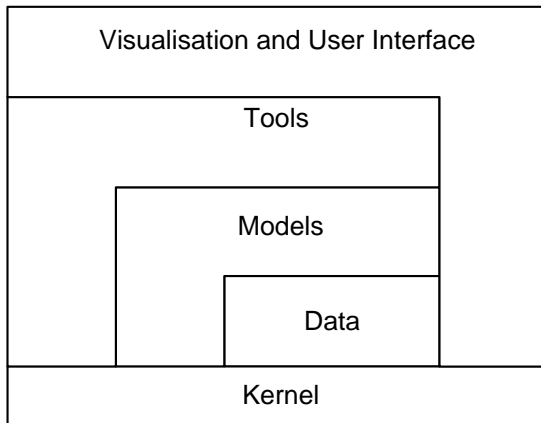


Figure 3: Overview of TIME architecture

The model makes use of TIME’s kernel, contained in the package `TIME.Core`. This package provides the parent class `Model`, along with the, optional, custom metadata tags (`Input`, `Output`, `Parameter`, `Minimum` and `Maximum`) that have been applied by the developer. The model code, and its parent class, do not include any reference to any of TIME’s support operations, such as data and model management, model linking, IO or visualisation. This independence, resulting from introspection, allows TIME models to remain stable even when the framework undergoes major revisions.

```

using System;
using TIME.Core;

public class SimpleModel : Model
{
    [Input]
    double Rainfall, PET;

    [Parameter, Minimum(0.0),
    Maximum(1.0)]
    double Coeff;

    [Output]
    double Runoff;

    public override void runTimeStep( )
    {
        Runoff =
            Coeff *
            Math.Max( 0.0, Rainfall - PET );
    }
}
  
```

Figure 4: Simple runoff coefficient model implemented in C# and TIME

4 MODEL PROCESSING TOOLS

Model introspection and metadata allows us to create a library of model processing tools. Examples of model processing tools include model schedulers, which control the timing and interaction of models, parameter space analysts, such as optimisers and model permutation tools, as well as user aids, such as user interface generators. Several of these tools are illustrated in ICMS and TIME while others are in development and planning.

In each case, introspection simplifies the development of the tool, while allowing for greater flexibility by keeping individual models decoupled from details of the framework and its tools. The presence of model processing tools, which make use of model metadata, also acts as an incentive to document a model, such as providing good variable names or metadata for numeric constraints.

4.1 Model Integration and Scheduling

Several protocols are available to integrate models, including the use of shared data with message passing [Watson et al. 2001] and dependency based systems that explicitly store the structure of communication paths between models [Reed et al. 1999]. Different integrated modelling applications may be more amenable to one approach over the other, making it important that modelling components, which may be reused in many applications, remain independent of model integration protocols.

ICMS uses introspection to provide a powerful model linking engine based on an explicit structure of model linkages. TIME currently has tools that support model integration using shared data. In both cases the integration protocol has had no impact on the structure of modelling components, allowing the frameworks to add support for the alternate protocols, or new protocols, without imposing a maintenance burden on model developers.

ICMS gives users a graphical canvas, shown in Figure 5 for connecting models and controlling flows of data. Because ICMS knows the various properties of a model, it can present the user with the available *ports* on an object and the linking engine can enforce rules such as “each link must be from an input to an output”.

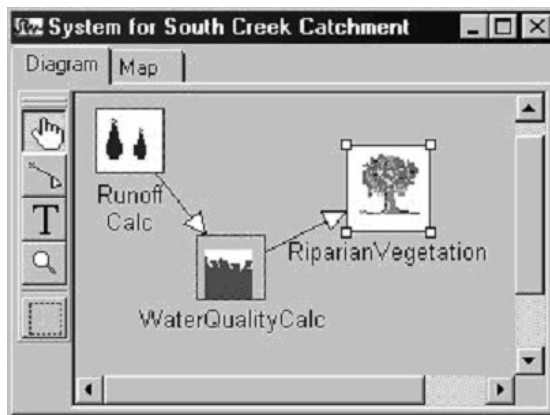


Figure 5: ICMS Model Linking Canvas

4.2 User Interface Generation

User interaction code, whether console based or graphical, typically consumes a significant amount of model development effort. Application and modelling frameworks typically streamline this job by providing specialised user interaction components, making the developer's job one of sensibly arranging these components for the user. In many cases, this arranging can be performed automatically using introspection, removing the need for the developer to write any user interaction code.

TIME uses introspection to provide both console and graphical interfaces to models, allowing developers to quickly modify and test components. Each property of the model is incorporated in the interface, with additional metadata, such as parameter constraints, being used to refine the appearance and behaviour of the interface. When models are run from a graphical interface, numeric properties are represented by slider bars, with controls for attaching input and output time series'. When a console interface is used, the system prompts the user for a value for each input, allowing time series' to be supplied for inputs or outputs. Figure 6 shows a graphical user interface (GUI), generated by TIME, for the model in Figure 4. Where numeric constraints have been supplied, such as $0 \leq Coeff \leq 1$, the slider bars have been appropriately constrained. As the model executes, slider bars representing model outputs and internal state variables move to dynamically reflect the results of the model.

TIME's user interface generation is intended to produce *technical* user interfaces suitable for use by researchers and developers, rather than high level interfaces to be used by non-technical stakeholders. These technical interfaces are very useful during

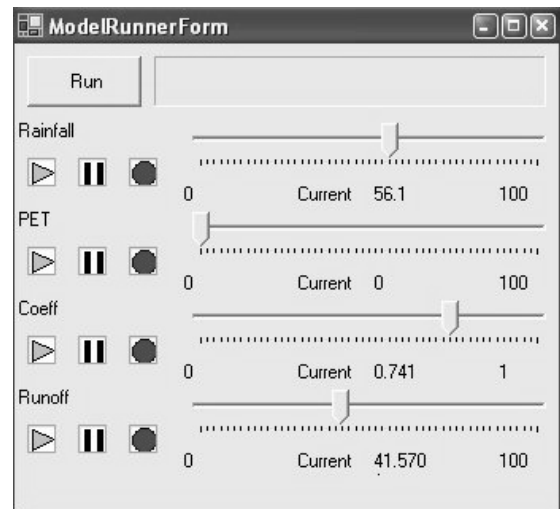


Figure 6: TIME Automatically generated user interface

model development because they remain consistent with changing model code. Furthermore they ensure the model code does not directly deal with any user interaction ensuring it is still possible to create polished user interfaces when a model is deployed in an application. By using any custom metadata to improve the generated GUI, TIME encourages the documentation of models early in the development cycle.

4.3 Time Traces

In a single cell model there are likely to be many variables that vary across time. In spatial models there will be many variables in each cell. Traditional model development has required model developers to provide explicit support for time traces. This typically results in either a rigid set of input and output time series' that must be supplied to a model, or complicated configuration code allowing the user to select individual properties. TIME uses introspection to allow a model developer to ignore most time varying inputs, while allowing the user to configure, at runtime, which inputs are *driven* by time series and which outputs are recorded.

Users are presented with an automatically generated user interface, where all numeric properties are initially configured as scalars (Figure 6). The user can select to *play* (▶) a time series into inputs, by dragging and dropping existing data, or *recording* (●) an output. Figure 7 shows the user interface for the runoff coefficient model (Figure 6) after the user has selected to play a rainfall time series and record a runoff time series.

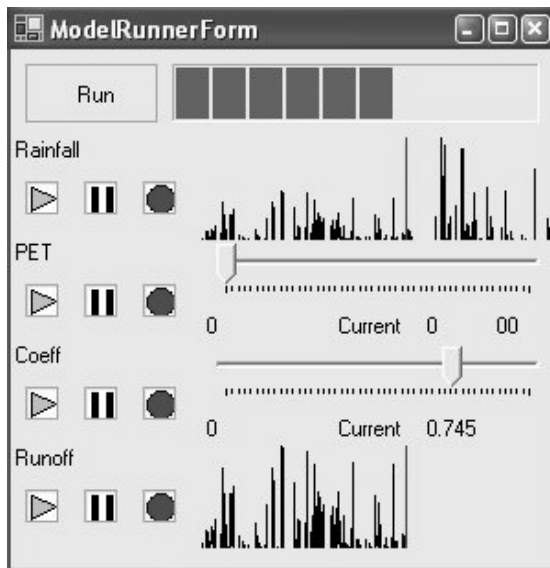


Figure 7: TIME Model with Time Series attached

Using introspection to handle time varying properties reduces the amount of code in a model, making it easier to develop and maintain models, while making the resulting models more flexible, by deferring decisions about how a model should be used, until it is. This flexibility to defer model configuration is particularly important when a model is used in a large integrated modelling exercise, where resource limitations require tradeoffs between what is recorded and what is discarded after each timestep.

4.4 Parameter Space Analysis

Several tools exercise a model's response over its parameter space. These include tools that find and optimise parameter sets for a given set of inputs and observed results, tools that permute over a region of the parameter space to map model sensitivity and tools that stochastically vary model parameters to simulate real world variability.

Model development often involves these, and similar tools, being custom developed for each model. Using introspection, each tool can be implemented using model metadata to discover names and ranges of parameters. A quasi-general optimisation component, based on direct search [Hookes and Jeeves 1961] is currently being developed in TIME.

4.5 Persistence

Model persistence is the ability to save a model's state (current values of all inputs, outputs and internal variables) at some time and load the model back

at a later time. It has traditionally fallen to the model developer to implement persistence mechanisms for a new model. This can be a time consuming process and typically results in persistence mechanisms that are highly version dependent, with new versions of models not reading older files.

ICMS can automatically save model configurations, including integrated models. TIME currently has support for loading the configuration of individual model components.

4.6 Model Distribution

Distributed Processing is the technique of sharing the computation of a system across multiple processors or computers. Distributed processing is either used because some resource is available only on certain computers or because the computational load of the system is too great for a single system. This second reason, parallel processing, is particularly attractive to users with large computational demands. Parallel processing becomes appropriate when many runs of a model are required, such as stochastic varying parameters, or when a very large model configuration can be partitioned into mostly independent submodels, such as different branches of a node-link network. Writing parallel processing applications from scratch can be complicated, with low level issues such as network communications and synchronisation requiring significant effort. By building upon other model processing tools, such as persistence and model linking engines, it is possible to create model distribution tools that remain transparent to the model developer.

5 PERFORMANCE CONSIDERATIONS

By dynamically examining components, introspection defers until runtime certain operations and checks that are typically performed at compile time. This deferral, while introducing great flexibility, introduces a runtime performance penalty. Understanding this penalty, and examining approaches to overcoming it, allows us to better design model processing tools.

Consider the hypothetical model configuration, shown in Figure 8, which incorporates several of the model processing tools discussed above. A model is configured to run 100 times, using a permutation tool that covers a region of the parameter space. For each iteration, the model is run for 10 years, on a daily timestep, with at least one daily time series input. Each timestep involves the performing

SomeOperation() over each one of 100 spatial units. An automatically generated user interface is used to dynamically reflect model state during this simulation. The markers A, B, C and D indicates the different levels of nesting where we might perform introspection. As might be expected, the greatest impact on runtime comes from code and operations residing in the inner loops C and D.

```
(A) For each Model Run (100 iterations)
... (B) For each timestep (3650 iterations)
..... (C) For each element (100 elements)
..... (D) SomeOperation( )
```

Figure 8: Model configuration using model processing tools

The permutation tool will configure the model, at marker B, before each of the 100 runs. The user interface is generated before the first run (at marker B) and updated for each timestep of each run (at marker C – 365000 times). Each time step the model is configured with the current value from the daily time series input (again at marker C – 365000 times).

Introspection operations performed in the outer loops, such as the permutation tool configuring model runs or the initial generation of the user interface, do not impose a noticeable performance penalty on the overall system. However, the introspection in the inner loops, required to drive the model with daily inputs and to dynamically update the user interface, can have a significant effect on runtime. In particular, when SomeOperation() is a trivial operation, such as the runoff coefficient calculation in Figure 4 and the model is aspatial, the time required for the housekeeping operations performed each timestep can be greater than the time to execute the model itself.

This typically isn't a problem, for a number of reasons. Firstly, the relative penalty reduces as the computational demands of the model increases, making it acceptable for most models. Secondly, much of the performance penalty derives from updating a user interface each timestep. For such dynamic visualisation to be useful, the model must be executing within the user's attention span, implying a reasonably short runtime, which introspection typically doesn't jeopardise. When a model is configured for a long run, or as part of a larger, integrated modelling application, a dynamic user interface showing each property of the model isn't typically used, eliminating most of the performance penalty. Finally, in specialised situations, where little, or no, performance penalty is acceptable, intro-

spection can be used to dynamically generate executable code, at some non time critical part of execution (such as marker A), that performs as well as explicit code.

The ability to dynamically generate wrappers, supported by .NET, allows introspection based frameworks to be used for developing, testing and debugging models, without incurring lasting speed penalties in deployed applications.

6 CONCLUSION

Model introspection and metadata have the potential to further reduce model development effort and make learning to develop within modelling frameworks easier. Self documenting components, using inferred metadata and custom metadata tags simplify the development of powerful model processors, including model linking engines and user interface generators.

When combined with other advantages of modern programming platforms, such as multi language development and web enabled models, introspection provides a powerful way forward for model developers and users.

REFERENCES

- Flanagan, D. *Java in a Nutshell*. O'Reilly and Associates, 3 edition, 1999.
- Hookes, R. and T. Jeeves. "Direct Search" solution of numerical and statistical problems. *J. Assoc. Comput. Mach.*, 8:212–229, 1961.
- Meyer, B. .NET is Coming. *Computer*, pages 92–97, August 2001.
- Reed, M., S. Cuddy, and A. Rizzoli. A framework for modelling multiple resource management issues – an open modelling approach. *Environmental Modelling and Software*, (14):503–509, 1999.
- Rizzoli, A., J. Davis, and D. Abel. Model and data integration and re-use in environmental decision support systems. *Decision Support Systems*, (24): 127–144, 1998.
- Watson, F., J. Rahman, and S. Seaton. Deploying environmental software using the Tarsier modelling environment. In *Proceedings of the Third Australian Stream Management Conference*, volume 2, pages 631–637, August 2001.