CSIRO

Developing Components for the Common Modelling Protocol

Version: October 2007

www.csiro.au

Developing Components for the Common Modelling Protocol

Contents

1. Introduction

This document is designed to assist builders of components that execute within the CSIRO Common Modelling Protocol (CMP). Topic areas covered are:

- an introduction to simulations;
- an introduction to CMP components;
- a description of the process by which components should be designed and implemented; and
- information about the infrastructure code provided by CPI to implement components.

Design of a component is properly the role of a “scientist”; the most important design questions relate to the way in which the various quantities and equations in a model should be structured. Implementation of a component, on the other hand, is the role of a “progammer”. Even if the same person carries out both design and implementation, it is useful to distinguish the two roles.

This document is based on the CMP implementation written by CSIRO Plant Industry (CPI). While the software development processes and examples described here are specific to the CPI implementation of the protocol, the “science” part of the component-building process should apply regardless of which implementation of the CMP is being used.

2. Simulations, models and sub-models

The purpose of any CMP component is to implement a *sub-model* that can be coupled with other sub-models to form a dynamic model. In order to develop components efficiently, a component designer must understand the concepts of *model, quantity, event* and *simulation*.

- A *simulation* is a computation of a *dynamic model* between given start and end times, i.e. it is an integration over time.
- A dynamic model is defined by a set of *equations*. The equations of a dynamic model may fall into natural groupings known as *sub-models*. Some of these sub-models may have equations and quantities of identical form, i.e. they belong to the same *sub-model class*. The dynamic model as a whole can therefore be viewed as a collection of instances of various sub-model classes.
- A sub-model is composed of a set of *quantities,* a set of *rate equations,* and a set of *events*.
- All quantities can be expressed as real numbers, integers, or Boolean values. (In the modelling protocol, these quantities can be organized into arrays and structures.)

  Real-valued quantities have *dimension* and *units*; the units must conform to the dimension. When designing a sub-model, it is useful to distinguish the following kinds of quantities:
  (a) *Constants* are quantities that are (i) invariant in time and (ii) have the same value in all instances of a sub-model within a model and all simulations of a model. For example, Avogadro's number is a constant.
  (b) *Parameters* are quantities that are invariant through the time course of a simulation, but may take different values between different instances of a sub-model within a model or between simulations of a model. For example, the radiation use efficiency of a plant species under reference conditions is treated as a parameter in many plant growth models.
  (c) *State variables* are quantities that may vary in time as the simulation is computed. The value of a state variable must be stored in order to compute the dynamics of the sub-model. An example is the water content of the surface 100mm of soil in a moisture budget. The initial value of each state variable must be specified in order for the simulation to be computed. There is a one-to-one correspondence between state variables and the rate equations of the sub-model. In principle, there should be no redundancy in the state variables.
  (d) *Summary variables* may also vary in time, but their value at any given time may be determined from the current values of the state and driving variables. Summary variables may be used to provide output from the simulation; to provide driving variables for other sub-models; or as notational conveniences in the specification of the sub-model's rate equations (in which case they may be referred to as "intermediate" variables). For example, in a model where the masses of different plant parts are followed the total shoot mass of the population is a summary variable.
  (e) *Driving variables* are quantities which are stored externally to a given sub-model but which must be known in order to compute the dynamics of the sub-model. They may (and usually do) vary in time. For example, minimum daily temperature and maximum daily temperature may be driving variables of a plant growth model.
- Each real-valued state variable has a *rate equation* associated with it. The rate equation is an ordinary differential equation that gives the rate of change of that state variable over time. The right-hand side of each rate equation must be composed only of constants, parameters, state variables, summary variables and driving variables belonging to the sub-model.
- Each submodel has zero or more *events*[1] associated with it. An event, in this sense, is
  - a set of equations defining an instantaneous change in one or more state variables; and

[1] N.B. The term "event" is used in another sense within the modelling protocol. Model events will be coded as protocol event handlers, but so will other model calculations. Unfortunately no good alternative term exists.

- a "trigger": a logical relation that, if satisfied at any time, causes the change(s) in state variables. Each event has zero or more quantities, known as *event parameters*, that may be used in specifying the right hand sides of the equations and the trigger along with constants, parameters, state variables, summary variables and driving variables belonging to the sub-model.

- A simulation is therefore completely defined by:
  - the model, i.e. the set of sub-models it contains;
  - the start and end times for the computation;
  - the values of the state variables and event parameters of each sub-model at the start time;
  - the time course of the model's driving variables.

3. A short introduction to the Common Modelling Protocol

3.1. Components and modules

Simulations that are computed within the CMP are constructed from *components*. Most components embody a sub-model.[2] An instance of a component within a simulation is known as a *module*. There may be many modules of the same component within a single simulation; for example, in a simulation with several paddocks there may be a water balance module for each paddock. Each module has a case-insensitive *name* (e.g. `water_b` in Figure 3.1).

3.2. Properties

Properties correspond to the quantities in a dynamic model. Every property forms part of a component. Each property is defined by a name; a type; and a value. Properties are referred to by their name: where necessary, the property name is qualified by the name of the module to which it belongs.

The type of a property determines the set of values it may take and the units of those values, where applicable. The type must be either one of a set of *primitive types* (usually real, integer, Boolean or text) or else an array or record structure ultimately composed of these primitive data types. The value must conform to the type. Type information is conveyed using an XML notation called Data Description Markup Language (DDML), which is described in section 3.7.

For example, in the simulation structure shown in Figure 3.1, `paddock_1.area` might have units of ha and a value of 150.0, while `paddock_2.area` would have the same units but a value of 350.0.

Two different kinds of properties are distinguished. *Driving properties* correspond to driving variables. The systems in a simulation work together to ensure that the values of each module's driving properties are read from the properties of other components as needed. A request for a driving property may result in zero, one, or more than one values being returned from different components, depending on the structure of the simulation.

All other properties (i.e. those that are stored by the component) are known as *owned properties*.

Some of the owned properties (usually those corresponding to state variables) have to be provided in order to define the starting state of a simulation: these are called *initialisation properties*.

3.3. Events and event handlers

Events in the common modelling protocol are used to signal the occurrence of activities (i.e. computations) and to pass instructions between components. Protocol "events" are therefore used to drive the computation of both continuous processes and logical "events" (discontinuous processes). Every event has a name; it may also have a type and parameter data that conform to the type. When an

Examples of valid units for real variables:

`g`	base (S.I.) unit
`hPa`	scaled unit
`MJ/m^2/d`	ratio with two terms in the denominator
`/s`	no numerator
`kg^0.75`	or `kg^.75`
`m^1/3`	but not `m^1/2`, which is grammatically correct but should be given as `m^0.5`
`g.m/s^2`	but not `m/s^2.g` as numerator terms must precede denominator terms

For more detail about how to specify units, see section 6.2 of the full CMP specification.

[2] Other components carry out utility tasks such as controlling the order of computations within each time step, the acquisition of the values of driving variables (for example weather data) or the storage of simulation results for later analysis. This document is primarily intended for developers of components that implement sub-models.

event is passed to a component, an *event handler* is executed.

3.4. Systems and simulations

A *system* is a component that groups related components within a simulation. In addition to the usual attributes of components, a system has zero or more components within it. The components in a simulation are therefore organized into a tree structure. At the root of the tree is the *simulation system*.

3.5. Messages

Messages are the means by which information and requests are passed between components and systems as a simulation is computed. Message data are transmitted in a binary format. There is a defined set of 31 messages. Every message has a common header that contains the message type (as a numeric index), the component from which it originated and the component to which it is to be sent. Most of the messages also contain further data, which are laid out in a specified fashion so that any protocol-compliant component can read them. A component that receives a message may execute some of its own internal logic (which may result in the component sending further messages); or it may be required to send particular messages as a mandatory response.

Component builders will not generally have to work with messages directly. A component API (implemented as a class) takes care of decoding incoming messages and encoding outgoing ones.

3.6. Identifiers for entities in a simulation

Any component, property or event can be uniquely identified within the simulation by its *fully-qualified name*. The fully-qualified name is constructed by adding the names of all containing systems and (where appropriate) components before the name of the entity, delimited with "`.`". For example, in Figure 3.1 the water balance component may have a property called `sw`; the fully-qualified name of this property in the `water_a` module would be `paddock_z1.water_a.sw`.

In addition to their names, all modules, properties and events within a simulation system have unique numeric identifiers. Each module is allocated a unique integer identifier during the initialisation of the simulation. Each property within a module has its own integer identifier, which is usually assigned by the component builder. The ordered pair [component ID,property ID] therefore uniquely identifies a property within the simulation. A similar scheme applies to event handlers.

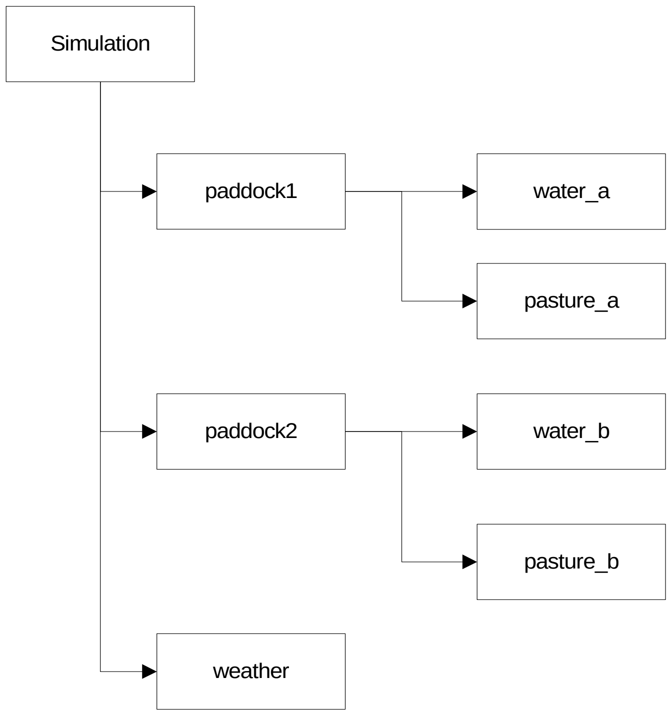


Figure 3.1. Components and systems in a CMP-compliant simulation.

3.7. Defining data types and units: DDML

Every property and event parameter has a type and (if a numeric real value) a unit. Within the protocol, information about types and units is given using Data Description Markup Language (DDML). DDML is an XML-based format; it is described in section 6 of the protocol specification document.

No knowledge of DDML is required to create properties that are scalars or arrays of scalars; the component-building API generates the DDML automatically for these types. For properties that contain multi-dimensional arrays or records, well-formed DDML descriptions of the type are required.

Examples of DDML definitions:

```
<type kind="double" unit="mm"/>             Double-precision scalar – note that the unit is given

<type kind="string" array="T"/>             Array of text strings

<type>                                      Record containing an integer, a real value and a logical value
  <field name="field_1" kind="integer4"/>
  <field kind="field_2" kind="double" unit="-"/>       Dimensionless value
  <field kind="field_3" kind="boolean"/>
</type>

<type array="T">                            Array of records....
  <element>                                 ... each record has a string and two arrays of numbers
    <field name="crop_ident"           kind="string"/>
    <field name="layers"     unit="mm" kind="double" array="T"/>
    <field name="uptake"     unit="mm" kind="double" array="T"/>
  </element>'
</type>
```

3.8. Simulation documents: SDML

Simulation Description Markup Language (SDML) is used within the protocol to specify the structure of a simulation and the initial values of the properties in each component. A valid SDML document is a complete specification of a single simulation. SDML is an XML-based format; it is described in section 7 of the protocol specification document.

Component developers will not generally need to handle SDML directly. The protocol implementation parses the simulation structure and creates the necessary modules automatically. The component-development API parses the initialisation data given in the SDML document and provides it to the component's code as *TTypedValue* objects (see section 5.3). See the document *Typed Value objects in the CSIRO implementation of the Common Modelling Protocol* for details on how to use these structured variables.

4. Designing a component: step by step

Adding a new model of biological or physical processes to the protocol typically follows three steps:

1. **Define the purpose & hence the "boundaries" of the sub-model.** The component designer needs to answer the following questions:
   - What quantities should the sub-model contain?
   - What quantities belong in other sub-models?
   - What calculations is the sub-model responsible for?
2. **Specify the interface of the sub-model.** Once a conceptual boundary has been drawn around a sub-model, it becomes possible to identify the flows of information that must take place as the sub-model is set up and its equations are computed. This information specifies the interface to the component. **We strongly recommend that the interface be formally specified** in a "component description document".
3. **Implement the sub-model as a component.** This process is described in sections 5 and 6.

These three steps perform a translation from a sub-model (a mathematical entity) to a component (a software entity). The roles required shift accordingly:

- Step 1 should be carried out by the "scientist" (the person responsible for the model equations).
- Step 2 should be carried out jointly by the "scientist" and the "programmer".
- Step 3 should be carried out by the "programmer".

In practice, of course, the component development process tends to be iterative; a component's interface evolves along with the scientific understanding that its sub-model embodies.

4.1. Purpose and boundaries

This step involves making an assessment of how the new processes relate to one another and to existing processes that are already modelled within other protocol-compliant components.

When a new sub-model is being added to a suite of existing components, its boundaries will generally be evident. Where an existing model is being re-implemented within the CMP, however, the component designer must decide whether to implement the model logic as one or as several components and where to set the boundaries between the new components.

The following points should be taken into account when delimiting component boundaries:

- The most important consideration is that equations that are tightly coupled (in the sense of sharing common quantities) should form part of the same component.
- Where a component is intended to be used in conjunction with other components, their conceptual boundaries should be aligned: no process should be represented in both components and all necessary processes must be represented in one of them.
- The component designer should aim to align component boundaries with those of existing components that perform the same or a similar function. This will facilitate component interchange.

Delimiting component boundaries: special cases

- Sometimes the question of the sub-model that should represent a given process is a matter of scientific debate, and a perfect alignment of component boundaries is therefore not feasible. (This usually happens at physical interfaces such as that between soil and roots.) In this case we suggest that a new component should be implemented so as to detect its companion components at run-time and to adjust its computations accordingly.
- Components with similar processes can have boundaries that align without being identical; two or more components may together carry out the functionality of a single alternative component. For example, in crop modelling the dynamics of dead surface residues are typically considered separately from that of green biomass, and the two sets of processes are implemented as separate components. In grasslands, however, both green and dead herbage are sources of animal feed and so it makes more sense to include them in a single component. To preserve alignment of the boundaries, the interface to the grassland component needs to correspond to that of both a crop and a surface residue component (except for the elements that communicate between the two).

- There is a design tradeoff to be made between having many small components, which will increase flexibility, and having few large components, which will increase computational efficiency. Also, designing components that are too small will make it difficult to configure simulations correctly; designing components that are too large will result in users having to define unnecessary initial values.

4.2. Identifying the submodel interface - properties

Once the conceptual boundary has been drawn around a sub-model, it becomes possible to identify the full set of quantities in the model equations. The different kinds of quantities will be handled differently in the component code:

Constants	Constants will generally be defined inside the model logic code and will not appear in the component's interface.
Parameters	Values for parameters will need to be provided at startup. The component will require either an *initialisation property* for each parameter, or else initialisation properties that allow the component to locate and read file(s) containing some or all of the parameters.
State variables	Initial values for state variables will need to be provided at startup. It is good practice to define an initialisation property for each state variable, so that the initial values for a simulation are recorded in the simulation's SDML document (see section 3.8). It is also good practice to make each state variable *readable*; code will need to be written to provide its current value to the rest of the simulation upon request.
Summary variables	Summary variables have different uses: • as outputs; • as driving properties by other components. In either case summary variables must be made readable if they are to carry out their function.
Driving variables	The *driving property* corresponding to each driving variable must be given a name and type that are the same as (or at least compatible with) those used in the component(s) that will provide values of the driving variable when simulations are run.

It is good practice to ensure that all initialisation properties are also readable; this simplifies the process for creating and restoring checkpoints.

The following points should be taken into account when defining the names and types of properties:

- One of the major benefits of the CMP is that it allows interchange of different models for the same or similar processes. This benefit will only be realised if properties with the same meaning are given the same definition. Component interface designers should therefore consult the documentation for related components and use common definitions wherever possible.
- Summary properties that are intended for use as outputs should be given a type and units that are convenient for users.
- Summary properties that are to be used as drivers for other components must be given a name and type that are the same as those used in the receiving component. (The CPI protocol implementation will carry out some type and unit conversions; see sections 8.4 and 6.2.3 of the full CMP specification for details.)

Writeable properties

Properties may be defined as being *writeable* as well as readable. Another component can request that a writeable property be reset to a different value.

Property resets should be used sparingly, if at all. As a rule, the only context in which they are defensible is when the period of scientific interest is after the start of the simulation and the resets are used to shift part of the system state to a desired point. Use of property resets in the regular computations of a component is a sign that its boundaries or events are badly defined.

4.3. Identifying the component interface - events

In the CMP, all computations within each time step are carried out in response to an event. When defining the component interface, it is therefore necessary to work out which equations should be computed together and so to group the model logic into *event handlers*.

Conceptually speaking, event handlers can be divided into three kinds:

- those which implement one or more rate equations and which should therefore be computed exactly once in each time step;
- those which implement the equations of an "event" in the sense of section 2, which may be computed zero or more times in each time step; and
- those that transmit information (for example the amount of a flow of organic matter).

The first kind are referred to as *sequenced event handlers*, because in the CPI protocol implementation the events that trigger them are usually generated by a special "sequencer" component that controls the computation of each time step. We recommend that sequenced event handlers be considered separately when documenting a component's interface.

The view within CPI is that information transmission should be carried out by means of property values, not events. The distinction between using an event to trigger a submodel "event" and using an event to transmit information is not a sharp one, however; even when an event is intended to trigger a computation, its parameters convey information to the component that receives it.

While the three kinds of event handlers are logically distinct, they are implemented in the same way within the component code.

The following points should be taken into account when defining the names and types of event handlers:

- As with properties, maintaining consistency between components (both those with similar event handlers and those that generate events with similar purpose) is an important design criterion.
- Where an event represents a management intervention, its parameters should be defined in terms that are meaningful to the user.

Computation order within time steps

Some components will need information that is computed by other components during the same time step. One example is the rate of some related process; another is the case where the source component reads time-varying information in from an external file. When this situation arises, it is important that computations take place only once the correct values of the necessary driving variables have been calculated.

In the CPI protocol implementation, this is generally done by specifying a "sequencing order" for each of the events that is triggered by the sequencer component

4.4. Component description documents

CPI recommends that the process of specifying the interface to a component should be formally documented in a fashion similar to that shown overleaf.

A sample component description document

This sample describes a real (if small) component:

1. Purpose of Component

The SOILT component encapsulates the EPIC soil temperature model (Williams 1985), as modified by Potter & Williams (1994).

2. Initialisation Properties

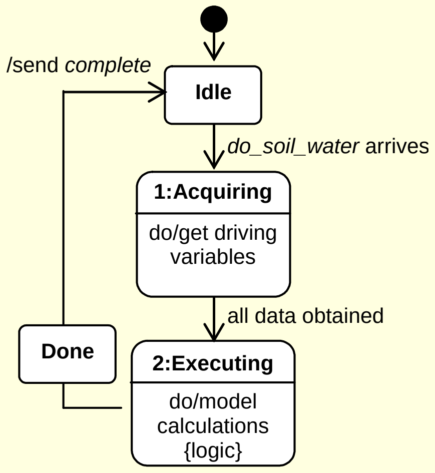
Property	Type	Units	Required?	Description
ann_temp	double	ºC	Yes	Long-term average annual temperature at the location of the simulation.
layers	double[ ]	mm	No	Thickness of each soil layer. If not provided, layer depths will be taken from the *bd_layer* driving property.
soilt	double[ ]	ºC	No	Initial soil temperature in each soil layer. If not provided, all layers will be initialised to the value of *ann_temp*.

3. Subscribed Events – Sequenced

3.1. *do_soil_water*

Default sequencing: 3000

Computes the soil temperature model for the time step.



4. Subscribed Events – Other

None.

5. Published Events

None.

6. Driving properties

Property	Type	Units	Event:State	Number[*]	Description
bd_layer	record		initialisation	1	Soil bulk density profile.
: *layers*	double[ ]	mm			
: *bd*	double[ ]	Mg/m^3			
cover_tot	double	m^2/m^2	*do_soil_temp*:1	0+	Total cover of co-occurring plant species.
daylength	double	hr	*do_soil_temp*:1	1	Day length including civil twilight.
maxt	double	ºC	*do_soil_temp*:1	1	Maximum air temperature.
mint	double	ºC	*do_soil_temp*:1	1	Minimum air temperature.
residue_cover	double	-	*do_soil_temp*:1	0+	Cover of standing dead and litter.
snow_pack	double	mm	*do_soil_temp*:1	0-1	Snow on the soil surface, in water equivalents. Default is 0.0.
sw_layer	record		*do_soil_temp*:1	1	
: *layers*	double[ ]	mm			Thickness of each soil layer.
: *value*	double[ ]	mm/mm			Current soil water content in each soil layer.
shoot_dm	double		*do_soil_temp*:1	0+	Total dry weight of all herbage.

[*] "Number" refers to the number of sources for the driving variable that is permitted.

A sample component description document (continued)

7. Owned properties

All initialisation properties are readable. In addition, the following owned properties are available:

(a) Standard properties

Property	Type	Units	Description
name	string		Fully-qualified name of the component.
type	string		Value is “Soil Temp”
version	string		Value is “1.0”
author	string		Value is “CSIRO Plant Industry”
active	Boolean		Denotes whether or not the component is active
state	string		SDML description of the current state

(b) Component-specific properties

Property	Type	Units	Description

n/a

Constructing and distributing such “component description documents” has significant benefits.

- It enables the “scientist” to provide the “programmer” with a clear account of the code that must be written in order to implement the component wrapper (see section 6).
- The descriptions of existing components are important resources for the developers of new components, because they provide the information about the kinds and types of properties and events that enable consistency of definitions to be achieved.

5. Implementation details

In the CSIRO PI (CPI) implementation of the Common Modelling Protocol, each model component is implemented as a class library in a Microsoft Windows DLL. The implementation of a model component simulation interface as computer code should be a routine task once the sub-model's interface is properly specified and documented and its equations are correctly coded.

5.1. Required binaries

Running simulations using the CMP requires a number of executable files in addition to the CMP-compliant components. The CPI protocol implementation structure is shown in Figure 5.1.

The central part of the CPI protocol implementation is `prot_cs.dll`. This DLL contains the *protocol engine*, which manages the messages that communicate between the modules in a simulation. The User Interface is built to allow the user to customise a simulation and display results. The `pisim.dll`, `timesvr.dll` and `sequencer.dll` components are supplied by CPI. They each carry out necessary tasks in the computation of a simulation, but they could be replaced by a component developer who wished to carry out their tasks in a different fashion. When using the dot net simulation engine (prot_cs.dll) it is not necessary to use any of the native Win32 libraries.

Module	***Description***
🟩 prot_cs.dll	Main processing engine implemented in a dot net library
🟩 native_int.dll	Used by prot_cs.dll to communicate with win32 native components. A mixed mode library.
🟩 piengineint.dll	Class library that allows a .net GUI to control either PI simulation engine.
🟩 cmpservices.dll	Class library that implements many useful classes such as XML parsing, structured TTypedValues, and base classes for logic components.
🟩 pisim.dll	The base simulation implemented in a component.
🟩 timesvr.dll	The component that is responsible for calculating the current time and when the end time is reached, terminates the simulation.
🟩 sequencer.dll	The component that publishes the events that occur regularly at every time step in the simulation.
🟧 prot.dll	Main processing engine (Win32 native)
🟧 protproxy.dll	Required if a .net GUI is to execute the native CMP engine.
🟧 piwrapper.dll	Required for use of win32 native PI components.

🟧 *Unmanaged native Win32 code*

🟩 *Managed .net code*

CMPServices class library is the key for component development. The base classes for a user's model logic are found in this library.

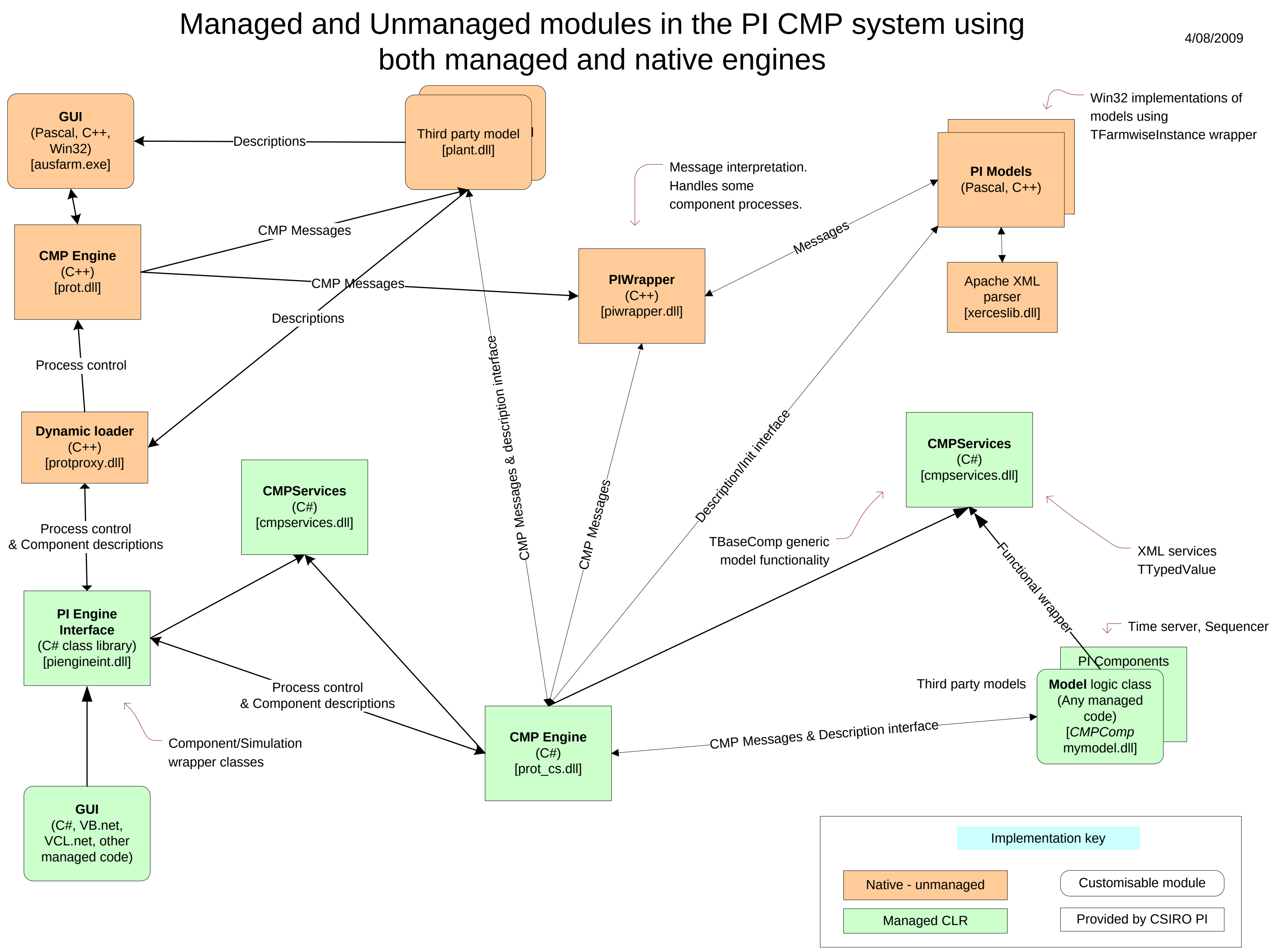

Managed and Unmanaged modules in the PI CMP system using both managed and native engines
4/08/2009
GUI (Pascal, C++, Win32) [ausfarm.exe]
Descriptions
Third party model [plant.dll]
Win32 implementations of models using TFarmwiseInstance wrapper
PI Models (Pascal, C++)
Message interpretation. Handles some component processes.
CMP Messages
CMP Engine (C++) [prot.dll]
CMP Messages
Descriptions
PIWrapper (C++) [piwrapper.dll]
Messages
Apache XML parser [xerceslib.dll]
Process control
Dynamic loader (C++) [protproxy.dll]
CMP Messages & description interface
CMP Messages
Description/Init interface
CMPServices (C#) [cmpservices.dll]
CMPServices (C#) [cmpservices.dll]
Process control & Component descriptions
TBaseComp generic model functionality
XML services TTypedValue
Functional wrapper
PI Engine Interface (C# class library) [piengineint.dll]
Time server, Sequencer
PI Components
Third party models
Model logic class (Any managed code) [CMPComp mymodel.dll]
Process control & Component descriptions
Component/Simulation wrapper classes
CMP Engine (C#) [prot_cs.dll]
CMP Messages & Description interface
GUI (C#, VB.net, VCL.net, other managed code)
Implementation key
Native - unmanaged
Customisable module
Managed CLR
Provided by CSIRO PI

Figure 5.1 Modules used in the PI CMP implementation

5.2. Component-building API: the **TBaseComp and TSysComp** *classes*

An efficient way to build CMP-compliant components, especially if they are to be executed within the CPI implementation, is to use CPI's component-building API. This API is provided as compiled code in a class library. The CMP engine communicates with the user's component via these classes. This allows the component builder to concentrate on coding only the sub-model-specific logic.

It is good practice to keep the system interface code and the code containing model logic separate. As a rule, a "logic class" containing the model equations will be written either prior to, or in parallel with, the implementation process described in section 6.

The main parent class in the component-building API is called *TBaseComp*. This class decodes and encodes messages and accesses initialisation information. The user's component code is exposed to the simulation engine through public functions defined in this class. Almost all the work in implementing a component is done by creating a descendant class of *TBaseComp* and overriding specific abstract and virtual functions.

Figure 5.2 illustrates the internal structure of two components implemented using the *TBaseComp* class. On the left, a *TTextReporter* logic class has been "wrapped" inside a *TBaseComp* descendant called *TTextOutInstance. TTextOutInstance* contains code to handle definition of the component properties and event handlers, initialisation, the acquisition of driving variables and provision of owned variables and the handling of events. On the right, a second logic class (*TLocalityModel*) has been "wrapped" inside a second *TBaseComp* descendant called *TWeatherInstance* that performs corresponding tasks.

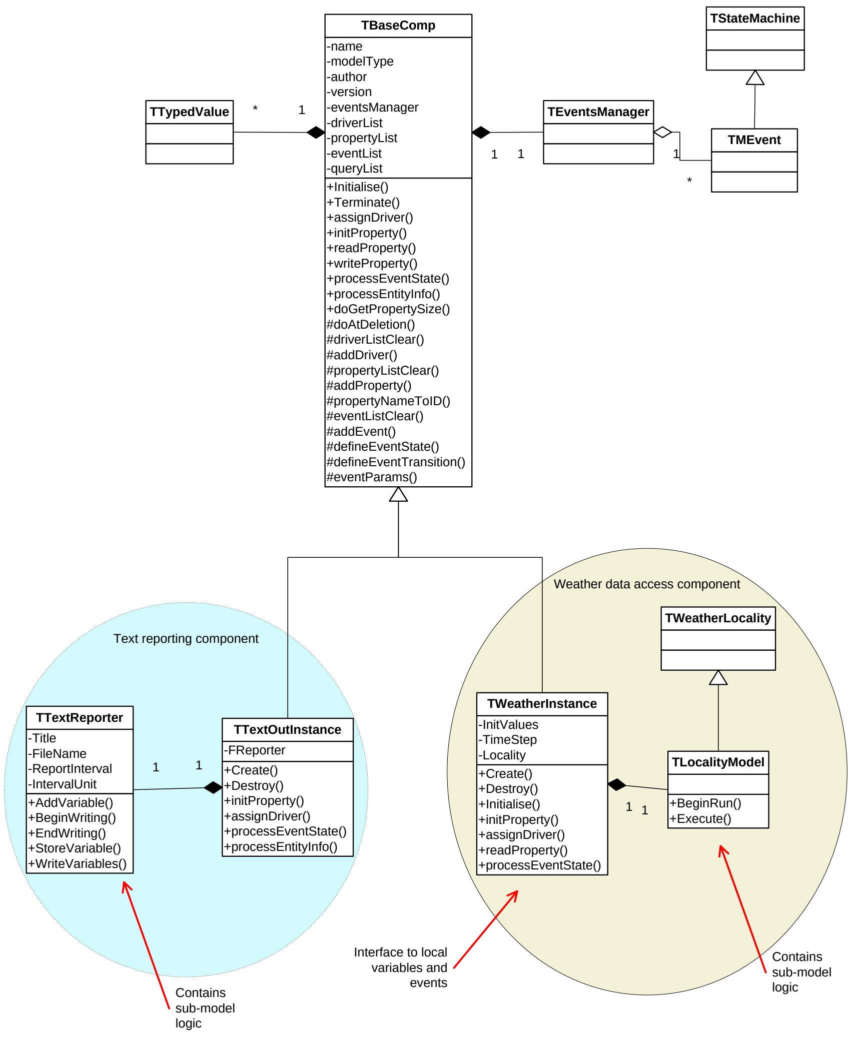


Figure 5.2. Classes involved in implementing two components by using *TBaseComp*.

5.3. Event processing using the state machine

The CMP implementation is designed to accommodate asynchronous processing. One consequence of this is that whenever an event handler generates a message that is sent to the rest of the simulation, the generating component must wait for the response to the message before executing the rest of the handler. In particular, this situation arises when the values of driving properties are requested: multiple properties can be requested at once, but the event handler must wait for all requests to be completed (and hence the driving values to be provided) before proceeding.

This requirement is met by implementing the code of each event handler within a *state machine*. The code of each event handler is divided into one or more *states*. The transitions from one state to another depend on the value of a *guard condition* that is set as each state is processed.

Figure 5.4 shows a common state machine for sequenced events, in which the values of driving properties are requested and then the event logic is computed. In this example, there is a single flow from state to state; in general different guard conditions can apply at the end of a state's processing, resulting in a state diagram with branching.

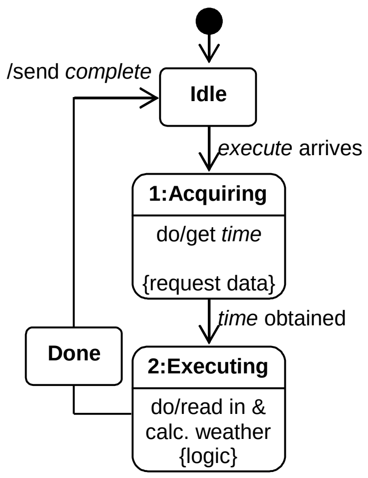


Figure 5.4. A simple state machine for an event handler.

6. Implementing a component: step by step

A component is implemented by creating a subclass from *TBaseComp* and overriding specific functions. When subclassing *TBaseComp*, the methods below must be implemented:

Constructor	This is the place where component-specific properties and events are defined. Add you custom code here to add properties and events required by your specialised component.
initialise*(int iStage)*	Called twice during the creation of each module. Stage will be either 1 or 2. See the *CMP Specification* document for a description of what happens at each of these stages. Add your custom initialisation code in this function.
initProperty*(int propertyID, TTypedValue aValue)*	Sets initial values for your custom component properties. This function is called by *TBaseComp* during init stage 1.
readProperty*(int propertyID, uint requestorID, ref TPropertyInfo aValue)*	Used to provide the current values of owned properties to the rest of the simulation This function is accessed when the component responds to a QueryValue message arriving due to another component issuing a GetValue message.
processEventState*(int iEventID, int iState, uint iPublisher, TTypedValue aParams)*	Handles events that arrive from other parts of the simulation. This is where the events and states defined earlier are processed from.

6.1. Defining the properties

Owned properties

In most cases, the properties and event handlers of a component are defined as part of the constructor for the descendant of *TBaseComp*. The *addProperty*() method is used to define each owned property's name, numeric ID, type and unit (where appropriate) and whether the property is readable, writable, and/or an initialisation property.

```
//addProperty(sName,    ID,       bRead, bWrite,bInit, sUnit,bIsArray, sType)
addProperty("start",    prpSTART, true, false, true,  "-", false, TTypedValue.STYPE_STR);
addProperty("finish",   prpEND,   true, false, true,  "-", false, TTypedValue.STYPE_STR);
addProperty("timestep", prpSLEN,  true, false, true,  "-", false, TTypedValue.STYPE_INT4);
addProperty("timeunit", prpSUNIT, true, false, true,  "-", false, TTypedValue.STYPE_STR);
addProperty("time",     prpTIME,  true, false, false, "-", false, TTimeStep.typeTIMESTEP);
addProperty("day",      prpDAY,   true, false, false, "-", false, TTypedValue.STYPE_INT4);
```

- The *ID* parameter is an integer. It is good practice to use constant values for property and event IDs to ensure consistency between the methods that use them.
- The *sType* parameter is either a valid DDML type description, in which case the *sUnit* and *bArray* parameters are ignored, or it is one of a set of predefined constants that denote a scalar type, in which case the *sUnit* and *bArray* parameters are used. The most common scalar type parameters are `TypeSpec.ITYPE_DOUBLE`, `TypeSpec.ITYPE_INT4`, `TypeSpec.ITYPE_STR` and `TypeSpec.ITYPE_BOOL`. A variety of other useful types are found in the TypeSpec class.
- Note that the first few property ID integer values (0-8) are reserved for standard properties possessed by all components. Component-specific property ID values should be equal to or greater than the constant value `PROP_START_INDEX`.
- Initialisation code in the initProperty() function (see below) must be provided for all properties for which the *bInit* parameter is set to TRUE.

Driving properties

The *addDriver*() method is used to define each driving property's name, numeric ID and type, plus the valid range for the number of responses received to a request for the driver. The three type-related parameters are handled in the same way as for owned properties:

```
//addDriver( sName,   ID,       iMinConn, iMaxConn,  sUnit, bIsArray, sType,          sourceID)
addDriver("time",     DRV_TIME, 1,         1,         "-",   false,  TTimeStep.typeTIMESTEP, 0);
```

6.2. Defining the event handlers

When defining each event handler, the parameters and the states and possible transitions in the state machine must be provided. The *addEvent*(), *defineEventState*() and *defineEventTransition*() methods are used for this purpose.

The following code defines the (sequenced) event handler shown in figure 5.4. A second, simpler event is also defined that does not require any driving property values; this state machine is typical for management events:

```
private const int evtEXEC = 1;
private const int evtSOMETHING = 2;
private const int smACQUIRE = 1;
private const int smPROCESS = 2;

private const string typePARAMS = "<type>"
             +   "<field name=\"param1\" kind=\"double\" unit=\"kg/ha\"/>"
             +   "<field name=\"param2\" kind=\"string\"/>"
             + "</type>";

:
:

// a simple sequenced event with driving property requests
addEvent( "do_process", evtEXEC, TypeSpec.KIND_SUBSCRIBEDEVENT, TypeSpec.TYPEEMPTY );
defineEventTransition(evtEXEC, TStateMachine.IDLE, 0, smEXECUTE, true);   //idle -> smExecute
defineEventState( evtEXEC, smACQUIRE, TStateMachine.NONLOGIC );     // eventID, stateID, type
defineEventState( evtEXEC, smPROCESS, TStateMachine.LOGIC    );
defineEventTransition( evtEXEC, smACQUIRE, FAIL, TStateMachine.DONE);     // eventID, stateID,
defineEventTransition( evtEXEC, smACQUIRE, 0,    smPROCESS );             //condition, toState
defineEventTransition( evtEXEC, smPROCESS, FAIL, TStateMachine.DONE);
defineEventTransition( evtEXEC, smPROCESS, 0,    TStateMachine.DONE);

// a simple management event
addEvent("do_something", evtSOMETHING, TypeSpec.KIND_SUBSCRIBEDEVENT, typePARAMS);
defineEventTransition(evtSOMETHING, TStateMachine.IDLE, 0, smPROCESS, true); //idle->smPROCESS
defineEventState(evtSOMETHING, smPROCESS, TStateMachine.LOGIC);
defineEventTransition(evtEXEC, smPROCESS, 0, TStateMachine.DONE);
```

- Note the use of `TypeSpec.TYPEEMPTY` for *sType* in the first event; sequenced events may not have parameters as the sequencer component has no way of determining values for them.
- "`LOGIC`" states do not contain any communication with the rest of the simulation, and so processing can move to the next state immediately. "`NONLOGIC`" states must wait for information for the rest of the simulation before processing continues. A state should be defined as "`NONLOGIC`" whenever it includes a property value request or a request for information about the structure of the simulation.

6.3. Initialising properties

When a simulation is executed, the initialisation process for each component results in three sets of method calls to the descendant of *TBaseComp*:

1. The ***initialise***() method is called with the *stage* parameter set to 1. Any component initialisation code that was not implemented in the constructor can be implemented here.

2. The ***initProperty***() method is called for each initial value provided in the simulation's SDML document. The implementation of *initProperty*() must store these value data, often by passing them to the model logic object.
3. Once all the initial value data have been passed, the ***initialise***() method is called a second time with *stage* parameter set to 2. Any component initialisation code that depends on combining initialisation values can be implemented here.

As a result, the *initialise*() and *initProperty*() implementations tend to look like this:

```
public override void initialise(int iStage)
{
   if (iStage == 1)
   {
       //my stage 1 code
   }
   if (iStage == 2)
   {
       //my stage 2 code
   }
}


public override void initProperty(int propertyID, TTypedValue aValue)
{
   string sText = aValue.asStr();

   switch (propertyID)
   {
      case prpSTART:
         {
            FSimRange.getStart().setFromAnsiDate(sText);
         }
         break;
      case prpEND:
         {
            FSimRange.getFinish().setFromAnsiDate(sText);
         }
         break;
      case prpSLEN:
         {
            FStepLength = aValue.asInt();
         }
         break;
      default:
      {
         sendError("My component: Invalid property ID in initProperty", true);
      }
      break;
   }
}
```

6.4. Acquiring values for driving properties

Driving properties must be defined before they can be requested (see section 6.1). Values of driving properties are requested within event handlers by calling the *sendDriverRequest*() method. Zero or more responses to the request may be received. A call to the virtual function *assignDriver*() will be made for each valid response; this method must therefore implement code to store or otherwise handle the driving variable's value.

A typical implementation of *assignDriver*() looks like this:

```
public override void assignDriver(int iDriverID, uint iProvider, TTypedValue aValue)
{
   switch (iDriverID)
   {
      case DRV_TIME:
         {
            int startDay = aValue.member("startDay").asInt();
            simulationTime.Set(startDay, 0, 0);
         }
         break;
      default:
         {  //not in this list so try a dynamically created driver
            base.assignDriver(iDriverID, iProvider, aValue);
         }
         break;
   }
}
```

6.5. Implementing model logic – event handlers

All event-handling code must be implemented within the *processEventState*() function of the descendant of *TBaseComp*. Code must be written for each state of each event. The function result must give a valid guard condition value for each state of each event (i.e. one that has been given in a call to *defineEventTransition*()) as it will determine the next transition of the state machine.

The following function might implement the two events that were defined in the example code in section 6.2:

```
public override int processEventState(int iEventID, int iState, uint iPublisher, TTypedValue
aParams)
{
    int iCondition = 0;         //guard condition to return

    // do_process event
    if (iEventID == evtEXEC)
    {
        switch (iState)
        {       //choose the state to execute
            case smACQUIRE:
                {
                    sendDriverRequest(drvTIME, iEventID);  // causes FTimeStep to be
                    iCondition = 0;                         // populated
                }
                break;
            case smPROCESS:
                {
                    FModel.Execute(FTimeStep.Start.iDay);
                    iCondition = 0;
                }
                break;
            default: iCondition = base.processEventState(iEventID, iState, iPublisher,
aParams);
        }
    }
    else
    {
        // do_something event
        if ((iEventID == evtSOMETHING) && (iState == smPROCESS))
        {
            FModel.doSomething(aParams.member("param1").asDouble(),
            aParams.member("param2").asStr());
            iCondition = 0;
        }
        else //when no events found to execute here always execute the base function
            iCondition = base.processEventState(iEventID, iState, iPublisher, aParams);
    }
    return iCondition;
}
```

Note the way in which:

- each state of each event has a block of code that sets the guard condition;
- information from the rest of the simulation (driving property values or event parameters) is passed to methods of the model logic class for processing;
- all event-state combinations that were not defined as part of the event's state machine are passed to the base class' version of *processEventState*()

6.6. Providing values for owned properties

Requests (components issuing GetValue messages) from the rest of the simulation for the value of an owned property will result in a call to the *readProperty*() method in the descendant of *TBaseComp*. Code must be written to populate the *TTypedValue* parameter of this method with the current value of the nominated property. The value parameter is guaranteed to be of the correct type, but the lengths of any arrays in the component's value must be set to the correct length as part of populating the value.

The *readProperty*() method will typically look something like this:

```
public override void readProperty(int propertyID, uint requestorID, ref TPropertyInfo aValue)
{
    switch (propertyID)
    {
        case prpSTART:
            {
                aValue.setValue(FSimRange.getStart().asDateTimeStr());
            }
            break;
        case prpEND:
            {
                aValue.setValue(FSimRange.getFinish().asDateTimeStr());
            }
            break;
        case prpSLEN:
            {
                aValue.setValue(FStepLength);
            }
            break;
        default:
            {
                string errorMsg = "MyComp: Invalid property ID: " + propertyID.ToString() + "
requested in readProperty()";
                sendError(errorMsg, true);
            }
            break;
    }

}
```

6.7. Publishing events

The process for sending an event from a component to the rest of the simulation (thereby triggering a computation in another component or components) is:

1. Define the name and type of the event to be published via a call to *defineEvent*() with the *kind* parameter set to `TypeSpec.KIND_PUBLISHEDEVENT`. This is usually done in the component's constructor.
2. Call the *sendPublishEvent*() when the event is to be triggered. This function takes the numeric ID of the event as an argument. If the event has parameters, they reside in a *TTypedValue* that can be obtained via the event item in the eventList[]. The event fields must be populated before the call to *sendPublishEvent*().

```
eventList[evtPROCESS].member("param1").setValue(1.0);
eventList[evtPROCESS].member("param2").setValue("something");
sendPublishEvent(evtPROCESS, true);   // eventID, bAcknowledge
```

For further discussion of published events, see section 2.3 of the CMP protocol specification document.

6.8. Handling errors

Components may report errors at any time during the simulation, or while the simulation is starting or terminating. Errors fall into two categories; fatal and non-fatal. Fatal errors will cause the simulation to terminate. Within a *TBaseComp* subclass, the *sendError*() method is used to report an error and, if necessary, to terminate the simulation. As a rule, therefore, exception-handling code should include a call to *sendError*().

```
try
{
   //...some code
}
catch (Exception e)
{
   String errorMsg = String.Format(FName + " myFunction(): {0}", e.Message);
   sendError(errorMsg, true);
}
```

6.9. Discovering the current time step

To discover the start and end times of the current time step, a component must define the `time` driving property. This property must have the `TTimeStep.typeTIMESTEP` type, which is a record with the following fields:

```
<type>
  <field name="startDay"     kind="integer4" unit="d" />
  <field name="startSec"     kind="integer4" unit="s" />
  <field name="startSecPart" kind="double"   unit="s" />
  <field name="endDay"       kind="integer4" unit="d" />
  <field name="endSec"       kind="integer4" unit="s" />
  <field name="endSecPart"   kind="double"   unit="s" />
</type>
```

The two day fields in this type are stored as Julian Day Numbers. *TTypedValue* data for this property can be decoded and manipulated using the *TTimeValue* class that is provided as part of the component API in the CMPServices assembly.

6.10. Setting values of other component's properties

Owned properties of other components that are writeable may be set using the RequestSet message from any other component. The basic process is:

1. Decide on the full name of the property you want to set.
2. Register a 'setter' property in your component.
3. Initialise the 'setter' property with a value.
4. Build and send a request set message (with ref to your new 'setter')

TBaseComp contains some helpful functions for this task.

```
newSetterName = childName + ".some_variable";
newSetterID = 0;

if (!getSetterByName(newSetterName, ref newSetterID))
{
     newSetterID = setPropertyList.Count;   //next id. Assume array index is the ID.
     string buf = "";
     MakeDDML("some_variable", TTypedValue.STYPE_STR, false, "", ref buf);
     //do the registration of the new setter
     addSetterProperty(newSetterName, newSetterID, newSetterName, 0, "", false, buf);
}

setPropertyList[newSetterID].destName = newSetterName; //store fqn & the dest state property
setPropertyList[newSetterID].regID = (uint)newSetterID;
setPropertyList[newSetterID].setValue("New value here!"); //store the value
msg = buildRequestSetMsg(newSetterID, setPropertyList[newSetterID]);

sendMessage(msg);
```

OR

```
string propName = "some_var";
string propFQName = aCompName + "." + propName;
string sDDML = "";
//utility function for creating a DDML type string - not always required
MakeDDML(propName, TTypedValue.STYPE_STR, false, "", ref sDDML);
TDDMLValue newValue = new TDDMLValue(sDDML, "");
newValue.setValue("New string value");

sendRequestSet(propFQName, sDDML, newValue);
```

The second option is more concise but does not allow the flexibility of accessing message properties before the message is sent.

7. Advanced topics

7.1. The connection scheme

When a simulation contains multiple modules drawn from the same component, the question arises: which of many possible sources for a driving property should be used? We refer to the rules for determining this as a *connection scheme*.

In the CPI protocol implementation, connections for a driving property are made to those source components that are equally-nearest to the requesting component. “Distance” is calculated using the number of steps up and down the hierarchical tree of components. Components cannot connect to themselves.

Whenever the simulation is restructured, the system components update the set of connections for each driving property belonging to components within their system.

7.2. Querying the simulation for its structure

The *sendQueryInfo*() method requests information about a component, property or event elsewhere in the simulation. Responses will result in a call to the *processEntityInfo*() method.

7.3. Other **TBaseComp** methods that generate messages

sendPauseSim()	Pauses or resumes a simulation.
sendActivateComp()	Activates another component.
sendEndSimulation()	Terminates the simulation.
sendComplete()	Sends an acknowledgement to another component. This method should only be called in a context mandated by the protocol specification.
addSetterProperty()	Adds a property reset request to the component and registers it.
sendReqSetValue()	Requests that the value of another component’s property be reset.
deleteProperty()	Removes an owned property from the component. It is then deregistered.
deleteDriver()	Removes a driving property from the component. It is then deregistered.
deleteSetterProperty()	Removes a property reset request from the component. It is then deregistered.
deleteEvent()	Removes an event from the component and deregisters it.

8. Starting the Component project

8.1. Structure

The component can be coded in any dot net language that will compile to a managed class library. Using Microsoft Visual Studio it is easy to begin a Class library project in C#. The CMPServices assembly must be included in the references. Also a file containing a class call *TComponentInstance* is required. This class is derived from your component class.

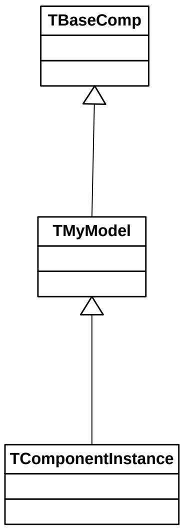


Figure 8.1 *Class hierarchy in the custom component.*

All the custom component code can exist in *TMyModel* and any helper classes you require. *TComponentInstance* is a stub that ensures the CMP engine can create an instance of your component class.

8.2. Template code

```
using System;
using System.Collections.Generic;
using System.Text;
using CMPServices;
using mynamespace;   //namespace of TMyModel


namespace CMPComp
{

    //=========================================================================
    /// <summary>
    /// Standard interface to the CMP engine. This class must be derived from the
    /// user's custom model class.
    /// </summary>
    //=========================================================================
    class TComponentInstance : TMyModel
    {
        public TComponentInstance(uint compID, uint parentCompID, MessageFromLogic
messageCallback)
            : base(compID, parentCompID, messageCallback)
        {

        }
        public void deleteInstance()
        {
        }

    }
}
```

The code following is an example of a model implemented in a class called *TMyModel*. It includes properties and events to accumulating day degree values.

```
using System;
using System.Collections.Generic;
using System.Text;
using CMPServices;

namespace mynamespace
{
    //==========================================================================
    /// <summary>
    /// Custom example component
    /// </summary>
    //==========================================================================
    public class TMyModel : TBaseComp
    {
        //set some instance specific information
        private static String _STYPE = "My Custom Model";
        private static String _SVERSION = "1.00";
        private static String _SAUTHOR = "Neville Herrmann";

        private const int PROP_1 = PROP_START_INDEX;

        private const int PROP_MAXT = 1; //driver

        //subscribed events
        private const int EVENT_COMPUTE = 1;
        private const int EVENT_RESET = 2;

        //event states
        private const int STATE_ACQUIRE = 1;
        private const int STATE_EXECUTE = 2;

        private double FComputedValue; //(PROP_1)
        private double FMaxTemp;

        //==========================================================================
        /// <summary>
        /// Constructor where all the custom properties and events will be defined.
        /// </summary>
        /// <param name="compID"></param>
        /// <param name="parentCompID"></param>
        /// <param name="messageCallback"></param>
        //==========================================================================
        public TMyModel(uint compID, uint parentCompID, MessageFromLogic messageCallback)
            : base(compID, parentCompID, messageCallback, _STYPE, _SVERSION, _SAUTHOR)
        {
            FModuleName = "template.dll";   //name this module

            //add properties and events
            TInitValue newProperty;
            newProperty = addProperty("day_degree", PROP_1, true, true, true, "-", false,
"double"); //==FComputedValue
            newProperty.setValue(0);
            addDriver("maxt", PROP_MAXT, 0, 1, "C", false, "double", 0);
            addEvent("compute", EVENT_COMPUTE, TypeSpec.KIND_SUBSCRIBEDEVENT,
TypeSpec.TYPEEMPTY, 0);
            defineEventState(EVENT_COMPUTE, STATE_ACQUIRE, TStateMachine.NONLOGIC);
            defineEventTransition(EVENT_COMPUTE, TStateMachine.IDLE, 0, STATE_ACQUIRE, true);
            defineEventState(EVENT_COMPUTE, STATE_EXECUTE, TStateMachine.LOGIC);
            defineEventTransition(EVENT_COMPUTE, STATE_ACQUIRE, 0, STATE_EXECUTE, false);
            defineEventTransition(EVENT_COMPUTE, STATE_EXECUTE, 0, TStateMachine.DONE, false);
//done

            addEvent("reset", EVENT_RESET, TypeSpec.KIND_SUBSCRIBEDEVENT, TypeSpec.TYPEEMPTY,
0);
            defineEventState(EVENT_RESET, 1, TStateMachine.LOGIC);
            defineEventTransition(EVENT_RESET, TStateMachine.IDLE, 0, 1, true);
            defineEventTransition(EVENT_RESET, 1, 0, TStateMachine.DONE, false); //done
        }
        //==========================================================================
        /// <summary>
        /// Sets up the first time step as part of processing "init1".
        /// </summary>
        /// <param name="iStage"></param>
        //==========================================================================
        public override void initialise(int iStage)
        {
            if (iStage == 1)
            {

            }
            if (iStage == 2)
            {

            }
        }
        //===========================================================================
        /// <summary>
        /// Initialise an owned property
        /// </summary>
        /// <param name="propertyID"></param>
        /// <param name="aValue"></param>
        //===========================================================================
        public override void initProperty(int propertyID, TTypedValue aValue)
        {
            switch (propertyID)
            {
                case PROP_1: FComputedValue = aValue.asDouble();
                    break;
            }
        }
        //===========================================================================
        /// <summary>
        /// Returns the current value of owned properties
        /// </summary>
        //===========================================================================
        public override void readProperty(int propertyID, uint requestorID, ref TPropertyInfo
aValue)
        {
            switch (propertyID)
            {
                case PROP_1: aValue.setValue(FComputedValue);
                    break;
            }
        }
        //===========================================================================
        /// <summary>
        /// Allows the simulation system to set the value of a local property. This must
        /// be implemented here if you want to set a writeable property.
        /// </summary>
        /// <param name="iPropertyID">Local ID of the property</param>
        /// <param name="aValue">New value as a TTypedValue</param>
        /// <returns>Returns true if the value could be set</returns>
        //===========================================================================
        public override bool writeProperty(int propertyID, TTypedValue aValue)
        {
            bool result = false;

            switch (propertyID)
            {
                case PROP_1:
                    FComputedValue = aValue.asDouble();
                    break;
                default:
                    result = false;
                    break;
            }

            if (!result)
            {
                return base.writeProperty(propertyID, aValue);
            }
            return result;
        }
        //===========================================================================
        /// <summary>
        /// Set the value of one of the driving variables
        /// </summary>
        //===========================================================================
        public override void assignDriver(int iDriverID, uint iProvider, TTypedValue aValue)
        {
            switch (iDriverID)
            {
                case PROP_MAXT:
                    FMaxTemp = aValue.asDouble();
                    break;

                default:
                    base.assignDriver(iDriverID, iProvider, aValue);
                    break;
            }
        }
        //=========================================================================
        /// <summary>
        /// Handles a returnInfo
        /// </summary>
        //=========================================================================
        public override void processEntityInfo(string sReqName, string sReturnName, uint
iOwnerID,
                                               uint iEntityID, int iKind, string sDDML)
        {

        }
        //=========================================================================
        /// <summary>
        /// Processes one state of an event
        /// </summary>
        /// <param name="iEventID"></param>
        /// <param name="iState"></param>
        /// <param name="iPublisher"></param>
        /// <param name="aParams"></param>
        /// <returns>The guard condition/result of the state's processing</returns>
        //=========================================================================
        public override int processEventState(int iEventID, int iState, uint iPublisher,
TTypedValue aParams)
        {
            int gdCondition = 0;

            if (iEventID == EVENT_COMPUTE)
            {  //compute the day degree sum event
                switch (iState)
                {
                    case STATE_ACQUIRE:
                        sendDriverRequest(PROP_MAXT, iEventID);     //firstly get the maxt
from weather
                        break;
                    case STATE_EXECUTE:
                        FComputedValue += FMaxTemp;                               //calculate
                        break;
                }
                gdCondition = 0;
            }
            else
                if (iEventID == EVENT_RESET)
                {      //reset the day degree sum event
                    FComputedValue = 0;
                }
                else
                    gdCondition = base.processEventState(iEventID, iState, iPublisher,
aParams);
            return gdCondition;
        }

    }
}
```

9. Further documentation

9.1. Common Modelling Protocol specification

The Common Modelling Protocol has been defined in detail in a public document entitled: ‘Specification of the CSIRO Common Modelling Protocol’. This document covers in detail; definitions, system tasks, messaging, properties and events, data types, SDML, and some implementation techniques.

9.2. Typed Value specification

Throughout the PI implementation of the CMP, *TTypedValue* and its descendant classes are used to contain structured types with their definitions and values. This technique allows for types of any complexity to be specified and manipulated.

The document “Typed Value objects in the CPI implementation of the Common Modelling Protocol.” contains details about the *TTypedValue* class.

9.3. Help file

Use the *CMPServices.chm* help file for detailed class specifications.