Regional Simulation Model (RSM)
Management Simulation Engine (MSE)

Version 2.2.9

Controllers

Documentation and User Manual

South Florida Water Management District
Office of Modeling
Model Development Division (4540)

February 15 2005

Abstract

This document provides descriptions and documentation of water-mover controllers implemented in the Management Simulation Engine (MSE) component of the Regional Simulation Model (RSM).

Contents

1 Introduction

The RSM consists of two interactive, primary components, the Hydrologic Simulation Engine (HSE) and the Management Simulation Engine (MSE). The HSE is a finite volume, coupled surface/groundwater/stream flow numerical model which includes structure flow equations for a wide variety of control structures, and implements efficient numerical solutions of conjunctive hydrological simulations [1]. The MSE is comprised of two primary subcomponents: a suite of low-level structure control algorithms which serve as flow regulators for individual structures, and, a set of high-level supervisory control functions which provide dynamic controller modification and coordination intended to facilitate regional control objectives. The high-level functions are collectively referred to as 'supervisors', and are documented separately [2]. This document describes the development, implementation, and use of the low-level structure control algorithms, otherwise referred to as 'controllers'.

In accordance with the modular, object-oriented design of the RSM, along with it's flexible and extensible input format specification employing the extensible markup language (XML), the design of controllers for the RSM consists of a base controller class with a uniform interface and specialized derived classes. Consistent with this object-hierarchical design it is important that the controllers which serve to regulate the watermover mass flows do not depend on physical characteristics of particular water control structures. In this way, the controller maintains the ability to regulate flow for any class of watermover. The controllers therefore also have a uniform interface to the watermovers, allowing for dynamic controller switching. A depiction of the overall MSE architecture is shown in figure 1.

At the lowest layer is the hydrological state information (Σ) computed by the HSE. This information includes water stages, flow values, rainfall, ET, hydrologic boundary conditions, or any other state variable used as input or computed as output by the HSE. All such variables are made available to the MSE and Assessors through the implementation of a uniform data monitor interface. The top level of the MSE is the supervisory layer. The function of a supervisor is to produce the supervisory control signal (μ) for a single, or collection of hydraulic structure controllers. Once the controllers have computed their respective control values (χ), these signals are applied as flow constraints to the structure watermovers in the HSE. Each watermover will compute a maximum flow capacity based on the hydrological state conditions and hydraulic transfer function of the structure. The resultant controlled flow will be some fraction of the currently available

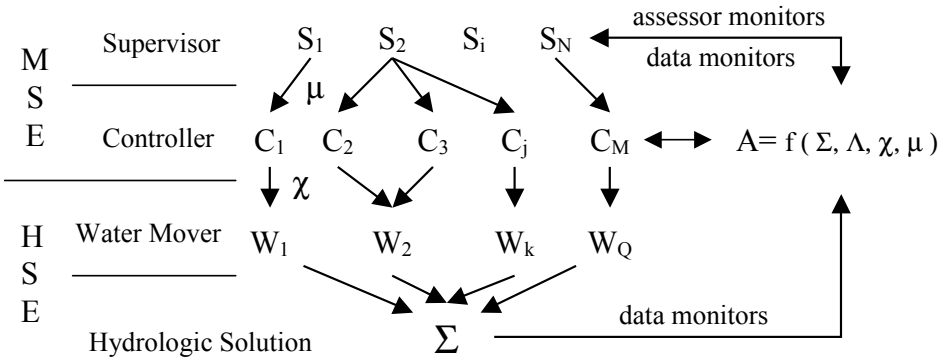

Figure 1: HSE MSE schematic

maximum flow capacity. Therefore, the design of the controller/watermover unitary interface depends upon output control signals in the interval of 0 to 1. A control of zero corresponds to no flow, a completely closed structure, and a control of one means the structure is fully open.

Important: The controllers are implemented as flow control regulators. The output of the controller is applied as an amplitude scale factor to the computed flow of the watermover. The intended range of controller outputs is in the interval of [0,1]. The user may define the control output range for all of the controllers except the Sigmoid controller. The user is *strongly cautioned* that implementation of controllers with output control ranges outside of the interval [0,1] may result in unintended modulation of watermover flow values.

1.1 Available Controllers & MSE Network

The current implementation of controllers include:

- PID controller
- PI with nonlinear activation (Sigmoid controller)
- Piecewise linear transfer function (SetPoint controller)
- Rule based expert system (Fuzzy controller)
- Finite state machine (User controller)
- Linear Programming interface (LP controller)

MSE controllers are defined and configured from within the `<controller>` sections of the RSM model XML file. Details on the allowable XML entries

are provided in each specific controller section.

An important feature of the MSE in general is the usage of Assessors and the MSE Network to provide synoptic state variable inputs for the controllers and supervisors. The MSE Network is an abstraction of the canal network and water control structures based on graph theory. It provides implicit aggregation of HSE canal network segments into Water Control Units (WCU's). The WCU is an integrated data object which stores assessed values of state variables relevant to the collection of HSE segments. Through the use of assessors and monitors, the controllers and supervisors access this synoptic information. The MSE Network also contains information relevant to management policies of the WCU's, and operational characteristics of the structure watermovers. The MSE Network is fully described in section 14.

1.2 Multiple Controllers

Implementation of generic, nonlinear field controllers, such as those commonly employed by the District, may be difficult to achieve with a single control algorithm per structure. The MSE controllers therefore support the notion of 'controller overloading'. This means that more than one controller may be attached to a watermover. The intention is that separate controllers with distinct control response characteristics, may be enabled in varying state-variable regimes suitable to each controller. For example, a simple piecewise linear transfer function may be used under dry conditions, while a rule based expert system (fuzzy) controller may be more effective in flood conditions. A MSE supervisor can select the controller for a watermover based on state-variable conditions [2].

Note that the MSE to HSE (controller to watermover interface) is not designed to allow multiple, concurrent controllers to provide flow modulation commands to a watermover. Although multiple controllers may be attached to a watermover, and can perform control algorithm computations concurrently (in the same timestep) only one controller per timestep will have it's control output applied to a watermover. It is the user's responsibility to ensure that multiple controllers are supervised accordingly. When multiple controllers per watermover are initially parsed and created from the XML model file specifications, the first controller that is parsed will be set as the active controller for the watermover. Subsequently, controllers that are encountered for the same watermover will replace the existing active watermover if the controller attribute `control="on"` is set.

2 Basic Control Theory

The development of control theory and its implementation is arguably one of the most influential technological achievements of man. Without the basis of negative feedback control, the industrial revolution would not have been possible. A brief review of control systems theory is given for convenience.

2.1 Open Loop Response

Consider a system as shown in Figure 2. The system is characterized by the *system function*, $H(s)$, which accepts inputs $X(s)$ and produces a corresponding output $Y(s)$.

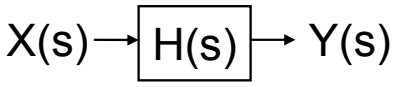

Figure 2: Open Loop System

The variable s represents a state variable of the system, it conventionally refers to the *Laplace* transform state representation of a time-domain signal, i.e.:

$$L(s) = \int_0^{+\infty} f(t)e^{-st}\,dt. \tag{1}$$

The response of the system is then characterized by the expression

$$Y(s) = X(s)H(s). \tag{2}$$

As the Laplace transform generalizes the Fourier transform (let $s = j\omega$), we can also think of $H(s)$ in terms of the system frequency response function. A system such as that shown in Figure 2 is an *open loop* system, there is no feedback of the output $Y(s)$ to the input of the system function. If the system function is stable and bounded in response to any system input, then the output is also stable and bounded.

2.2 Closed Loop Feedback

Consider now a simple extension to the open loop system depicted in Figure 3. In this scenario the output signal $Y(s)$ is fed-back to the input of the system, resulting in a *closed loop* feedback path. The feedback signal may be conditioned by a *feedback function* $F(s)$. The system function is preceded by a *control function* $H_c(s)$ that receives the combined input/feedback signal. The control function serves to adjust the system function input to achieve a desired system output.

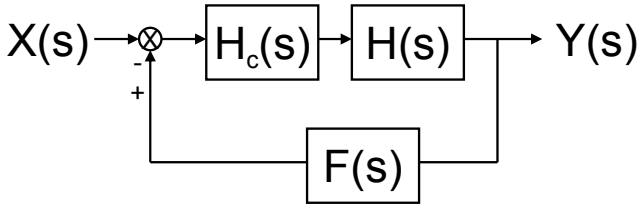

Figure 3: Closed Loop System

The desired system output is predetermined, and some distance metric between the desired output and the actual output defines an *error* signal. This error signal is presented as input to the control function. Minimization of the error signal defines the objective of the control function.

Consolidating the control function and system function into a composite system function $H(s)$, the modified system response is:

$$Y(s) = \frac{H(s)}{1 + F(s)H(s)} X(s). \tag{3}$$

One can immediately see that if the control function is not judiciously selected, the possibility exists that the denominator of system response can become zero, in which case the system output is unbounded (a pole), or that it could become uncontrollably large, resulting in zero system output (a zero). Thus a price that has to paid for increasing the system complexity towards control of the system response, is careful design and implementation of the control function.

3 Lookup Tables

As a simple alternative to actively controlling a rated flow or hydraulic structure watermover, the collection of lookup table watermovers can be used directly to simulate known flow characteristics. A simple example is the use of a 1-D lookup table to operate pumps when the water level in an agricultural area rises above a certain value. The following example shows a `<single_control>` watermover defined to discharge water from waterbody 11 into waterbody 12 based on stage levels in waterbody 21. Below stage levels of 12.5 there is no discharge. Between stage 12.5 and 13.5 a pump with discharge capacity of 400 units/s is activated. In the stage region of 13.5 - 14.0 a linear interpolation between 400 and 800 units/s is simulated, and above stages of 14, the discharge is constant at 800 units/s.

```
<watermovers>
  <single_control wb1="11" wb2="12" control="21" name="Smith farm">
      0.0   0
      12.5  400
      13.5  400
      14.0  800
      50.0  800
  </single_control>
</watermovers>
```

In this manner, a variety of control procedures can be simulated using `<single_control>`, `<dual_control>` and `<delta_control>` watermovers. Details on each of these lookup table watermovers is provided in the HSE User's Manual [5].

4 PID Controller

A significant development in the implementation of closed loop feedback control is the Proportional Integral Derivative (PID) control function. It essentially is comprised of a two-stage control function, a Proportional Integral (PI) control, in conjunction with a Proportional Derivative (PD) control. The PI control makes adjustments in response to the total (integral) of the error, and therefore serves to provide adjustments aimed at satisfying the steady-state system response. Without PI control, the system would not be likely to settle at a desired output value. The PD control supplies adjustments in response to changes in the system error, and therefore addresses the transient response of the system.

The addition of proportional, derivative and integral control functions can be represented as an extension of the system shown in Figure 3, where the control function is decomposed into the three components as shown in Figure 4.

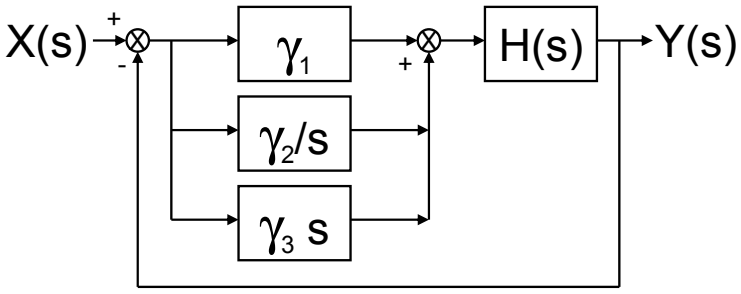

Figure 4: PID Control System

The corresponding control function for this implementation is

$$H_c(s) = \gamma_1 + \frac{\gamma_2}{s} + \gamma_3\, s \;\; = \gamma_3 \left(\frac{s^2 + \frac{\gamma_1}{\gamma_3}\, s + \frac{\gamma_2}{\gamma_3}}{s} \right). \tag{4}$$

Inspection of this expression reveals that this system function has two zeros and a pole. Again, care must be taken to ensure system stability in response to the implemented control function.

4.1 PID Implementation

In the time-domain, the PID control can be represented *via* the expression

$$h_c(t) = \gamma_P\, \epsilon(t) + \gamma_D \frac{d\epsilon}{dt} + \gamma_I \int_0^t \epsilon(t)\, dt. \tag{5}$$

where γ_P γ_D and γ_I represent gain factors for the proportional, derivative and integral terms, and ϵ the system error. Conversion of this expression into a time difference equation results in

$$h_c(i) = \gamma_P\, \epsilon_i + \gamma_D \frac{\triangle \epsilon_i}{\triangle t} + \gamma_I \sum_{i=1}^{n} \epsilon_i \triangle t. \tag{6}$$

Assuming that a simple arithmetic difference is employed as the system state error-metric:

$$\epsilon(t) = \phi(t) - T(t) \tag{7}$$

where the current system state variable to be controlled is $\phi(t)$ and the desired system target state $T(t)$, the control computation for a single time-step can be expressed as

$$h_c(i) = \gamma_P\, (\phi_i - T_i) + \gamma_D \frac{(\phi_i - T_i) - (\phi_{i-1} - T_{i-1})}{(t_i - t_{i-1})} + \gamma_I \sum_{i=1}^{n} (\phi_i - T_i)(t_i - t_{i-1}). \tag{8}$$

4.1.1 PID Tuning

Selection of the appropriate control gains is highly implementation dependent. The analytical solution is to investigate the locations of the poles and zeros in the s-plane applied to the root-locus methods. While this is a satisfying analytical exercise, it requires knowledge of the compensated system function $\frac{H(s)}{1+F(s)H(s)}$. When this is not feasible, various prescriptive algorithms such as the Zeigler-Nichols tuning method may be applied.

The Zeigler Nichols tuning method may be employed as follows:

1. Set the integral gain term to zero.
2. Gradually increase the proportional gain from zero until the system just begins to oscillate continuously. The proportional gain at this point is the ultimate gain, P_U. The period of oscillation at this point is the ultimate period, T_U. The ultimate period must be expressed in the same units used internally in the PID algorithm for calculations involving the time step. RSM uses seconds.
3. Set the proportional and integral gain values according to values in Table 4.1.1.

Control Type	Performance	γ_P	γ_I	γ_D
P	1/4 Decay	0.5 P_U		
PI	1/4 Decay	0.45 P_U	0.54 P_U/T_U	
PID	1/4 Decay	0.6 P_U	1.2 P_U/T_U	0.075 P_UT_U
PID	Some overshoot	0.33 P_U	0.66 P_U/T_U	0.11 P_UT_U
PID	No overshoot	0.2 P_U	0.606 P_U/T_U	0.10 P_UT_U

Table 4.1.1. Zeigler Nichols Tuning Parameters

4.2 PID XML

To implement a controller in the RSM, an appropriate controller definition must be provided for the respective watermover which is to be controlled.

The controller environments available for the PID controller are shown in Table 4.2.

environment	attribute	meaning
`<pidctrl>`		PID controller definition
	cid	positive (cid>0) controller id
	label	optional controller label
	wmID	ID of watermover to be controlled
	control	'on' or 'off'
	type	'positive' or 'negative' control
	offset	target value bias term
	nvals	number of integration values
	Gp	Proportional gain
	Gi	Integral gain
	Gd	Derivative gain
	ctrlMin	Minimum control output value
	ctrlMax	Maximum control output value
`<target>`		Target state specification
`<*monitor>`		State variable specification

Table 4.2. PID Controller XML

The `control` attribute can be used to deactivate the controller. If the value of `control` is set to any value other than "on", the controller will be deactivated. This means that the control output will be forced to a value of 1, no control output variations will occur. Since the control outputs are applied as amplitude modulation factors to the watermover flow, the watermover flow will default to it's uncontrolled values.

The `type` attribute controls the sign of the output control values. If `type` is "positive" (the default) then the output control values are computed according to equation (8). If `type` is "negative", the control output is the negative of equation (8).

The `offset` attribute allows the user to specify a numerical bias term which is applied to the target value in computation of the current controller

error term. If the `offset` value is set, the control deviation will be computed as:

$$\epsilon(t) = \phi(t) - [T(t) + \Omega] \tag{9}$$

where Ω is the value of the offset.

The `nvals` attribute is currently not used.

The following example illustrates a PID controller applied to a water-mover.

```
<controller id="1">
  <!-- Controller for discharge from segment 1 -->
  <pidctrl cid="101" label="PIDCtrl 1: " wmID="1"
   type="positive"
   nvals="10" Gi="0.01" Gd="0.0" Gp="7.0"
   ctrlMin="0.0" ctrlMax="1.0" >
    <target label="const_target">
       <const value="500.0"></const>
    </target>
    <segmentmonitor id="1" attr="head"></segmentmonitor>
  </pidctrl>
</controller>
```

A full example of PID controllers applied in a RSM test case is shown in 15.3.

5 Sigmoid Controller

Owing to the generic abstraction of a watermover as an object that is independent of physical constraints, a watermover may have no physical control variable which one can control to regulate flow. In such cases the controller may be implemented as a mass flow regulator which modulates the flow volume by a smooth, monotonic function in the interval [0,1]. To achieve this objective, the model development division has conceived and implemented a *Sigmoid controller*. The sigmoid (s-shaped) controller provides the desired functionality for watermovers without an obvious physical control variable. In other cases where a smooth control function is required with arbitrary state and control variable inputs, the sigmoid controller can provide the desired response and flexibility.

5.1 Sigmoid Implementation

The sigmoid controller is essentially a PI controller with a single nonlinear activation function (the sigmoid) filtering the controller output. The PI portion of the controller is implemented as follows

$$h_{PI}(i) = \gamma_P \, \epsilon_i + \gamma_I \sum_{i=1}^{n} \epsilon_i \triangle t. \tag{10}$$

Once a preliminary control output is available, the output is processed by a nonlinear sigmoidal activation function $S(x)$, the logistic function. This function serves to limit the possibly unbounded control outputs to the interval [0,1], while also providing an adjustable derivative for the linear portion of the activation function. Finally, the processed control signal is scaled by a constant scale factor α. The resultant sigmoid control signal is therefore given by

$$h_S(i) = \alpha \, S\left(h_{PI}(i)\right) \tag{11}$$

The logistic function is a special case of the more general hyperbolic tangent function. The hyperbolic tangent may be expressed in exponential terms as

$$\tanh(cx) = \frac{e^{cx} - e^{-cx}}{e^{cx} + e^{-cx}}. \tag{12}$$

As a result of the exponential basis of the hyperbolic tangent, its derivatives have a particularly simple expression. A problem with the hyperbolic

tangent for output control functions is that it has poles spaced at periodic intervals across its ordinal domain. The logistic function is however a bounded function with limits at $-\infty$ and $+\infty$ of 0 and 1 respectively. The logistic function is expressed as

$$S(cx) = \frac{1}{1 + e^{-cx}}. \tag{13}$$

with $c > 0$. The derivative is specified by $S'(cx) = c\,S(1 - S) > 0$, from which it follows that S is a smoothly increasing monotonic function. A plot of S(x) is shown in Figure 5 for several values of the positive constant c. The value of c determines the slope of the function at the origin, and can change the functional behavior from that of a slowly rising transition (c→0) to one of a unit step function ($c \to \infty$).

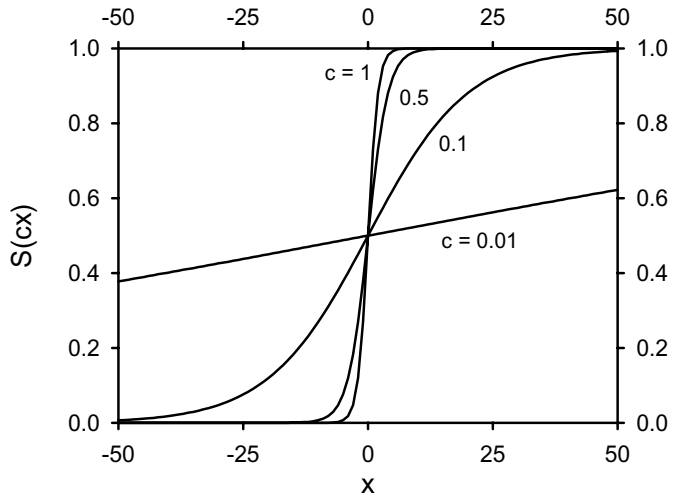

Figure 5: Logistic Activation Function

Variants of the hyperbolic tangent, or logistic sigmoidal functions are commonly employed as the neuronal activation function in neural networks. The sigmoid controller is therefore a PI controller with a single output neuron modulating the control function.

5.2 Sigmoid XML

The controller environments available for the Sigmoid controller are shown in Table 5.2.

environment	attribute	meaning
`<sigmoidctrl>`		Sigmoid controller definition
	cid	positive (cid>0) controller id
	label	optional controller label
	wmID	ID of watermover to be controlled
	control	'on' or 'off'
	type	'positive' or 'negative' control
	nvals	number of integration values
	offset	target value bias term
	scale	output scale factor α
	c	argument for exponential term
	Gp	Proportional gain
	Gi	Integral gain
`<target>`		Target state specification
`<*monitor>`		State variable specification

Table 5.2. Sigmoid Controller XML

The `control` attribute can be used to deactivate the controller. If the value of `control` is set to any value other than "on", the controller will be deactivated. This means that the control output will be forced to a value of 1, no control output variations will occur. Since the control outputs are applied as amplitude modulation factors to the watermover flow, the watermover flow will default to it's uncontrolled values.

The `type` attribute controls the sign of the output control values. If `type` is "positive" (the default) then the output control values are computed according to equations (11) and (13). If the `type` is "negative", the control output is computed by the equation

$$h_S(i) = \alpha \left[1 - S\left(h_{PI}(i)\right)\right] \tag{14}$$

The `offset` attribute allows the user to specify a numerical bias term which is applied to the target value in computation of the current controller error term. If the `offset` value is set, the control deviation will be computed

according to equation (9).

The `c` attribute specifies the numeric value of the exponential argument in the logistic function, equation (13).

The `scale` attribute sets the numeric value of the amplitude scale factor, α, in equation (11).

The `nvals` attribute is currently not used.

The following example illustrates a Sigmoid controller applied to a watermover.

```
<controller id="1">
  <!-- Controller for discharge from segment 1 -->
  <sigmoidctrl cid="101" label="SigmoidCtrl 1: " wmID="1"
   type="positive" control="on" c="0.1" Gp="10.0" Gi="0.4">
    <target label="const_target">
      <const value="500.0"></const>
    </target>
    <segmentmonitor id="1" attr="head"></segmentmonitor>
  </sigmoidctrl>
</controller>
```

A full example of the Sigmoid controller applied in a RSM test case is shown in 15.4.

6 SetPoint Controller

The PID and Sigmoid controllers are active-feedback, closed-loop response functions which seek to minimize the error between the actual system states and the desired system states. Maintenance of such controllers requires selection of gains and other parameters, which may not always be obvious or convenient. It is also realized that water management operational techniques have been derived over the years based on simple rules and binary or linear switches applied at known threshold points. The purpose of the setpoint controller is to provide such operational functionality.

6.1 SetPoint Implementation

6.1.1 Constant SetPoint

In its' simplest form, the setpoint controller simply assigns a constant control output value (the setpoint), regardless of the input state variables. The prescription for the controller output is therefore:

$$h_{SP}(i) = C_{sp}$$

where C_{sp} is the constant setpoint value specified for the controller.

6.1.2 Binary SetPoint

Now consider the case where a simply binary switch is desired, such that the controller output can assume only one of two values, based on the relation of a monitored state variable to a threshold (or trigger) value. A depiction of such a control transfer function is shown in Figure 6, where SP_L and SP_H represent the low and high set point values of the control output (h_{SP}), and τ_L and τ_H represent the low and high trigger values of the state variable ϕ which control the transition from SP_L to SP_H.

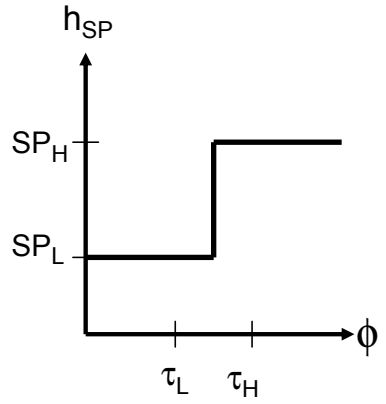

Figure 6: Binary Switch Set Point Control

This function is implemented in the RSM as follows:

$$h_{SP}(i) = \begin{cases} SP_L\ ; & \phi < \tau_L \\ SP_L\ ; & if\ (\phi - \tau_L) < (\tau_H - \phi)\ for\ \tau_L < \phi < \tau_H \\ SP_H\ ; & \phi > \tau_H \\ SP_H\ ; & if\ (\phi - \tau_L) >= (\tau_H - \phi)\ for\ \tau_L < \phi < \tau_H \end{cases} \tag{15}$$

It simply means that if the state variable is less than the lower trigger threshold, the lower setpoint value is returned. If the state variable is greater than the higher trigger threshold, then the higher setpoint value is returned. If the state variable is in the region between the low and high trigger thresholds, then whichever setpoint threshold which is closest to the state variable will be used.

The preceding binary setpoint assumed that the desired control transfer function was implemented as a positive (up) step function. The setpoint controller uses the `step` token to control the response of the transfer functions as either "up" or "down" progressing functions. In the case that the binary switch setpoint is implemented as a "down" step, the control transfer function is shown in Figure 7 with the following implementation is used:

$$h_{SP}(i) = \begin{cases} SP_H \ ; & \phi < \tau_L \\ SP_H \ ; & if \ (\phi - \tau_L) < (\tau_H - \phi) \ for \ \tau_L < \phi < \tau_H \\ SP_L \ ; & \phi > \tau_H \\ SP_L \ ; & if \ (\phi - \tau_L) >= (\tau_H - \phi) \ for \ \tau_L < \phi < \tau_H \end{cases} \tag{16}$$

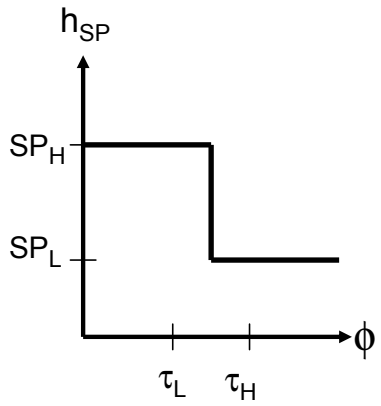

Figure 7: Binary Switch Set Point Control - Down Step

6.1.3 Linear Interpolated SetPoint

In situations where a linear interpolation between two setpoint values is appropriate, the setpoint controller will perform a linear extrapolation between the low and high setpoint abscissa and low and high trigger ordinates. A graphical depiction of this behavior is shown in Figure 8

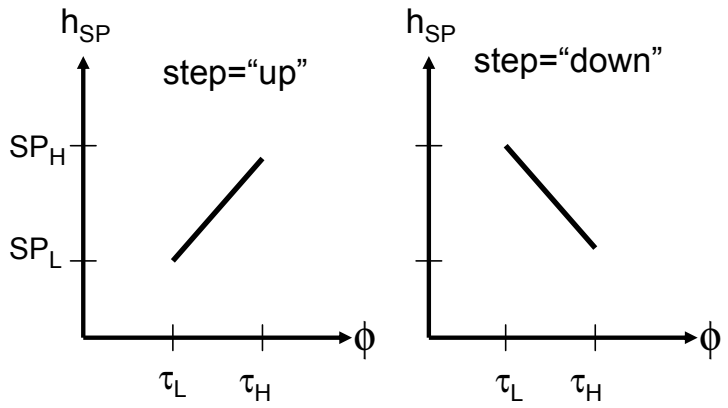

Figure 8: Linear Interpolation Set Point Control

The corresponding RSM implementation is prescribed as:

$$h_{SP}(i) = \begin{cases} C_{sp} \ ; & \phi < \tau_L \ or \ \phi > \tau_H \\ SP_L + \left(\frac{SP_H - SP_L}{\tau_L - \tau_H}\right)(\phi - \tau_L) \ ; & \tau_L < \phi < \tau_H \ \text{“}up\text{”} \\ SP_H - \left(\frac{SP_H - SP_L}{\tau_L - \tau_H}\right)(\phi - \tau_L) \ ; & \tau_L < \phi < \tau_H \ \text{“}down\text{”} \end{cases} \tag{17}$$

In this mode, if the state variable falls outside the trigger window, then the constant set point value is returned.

6.1.4 Binary Switch Plus Linear Interpolated SetPoint

It is possible to combine the behaviors of the binary switch and linear interpolation to achieve a composite setpoint control function as shown in Figure 9.

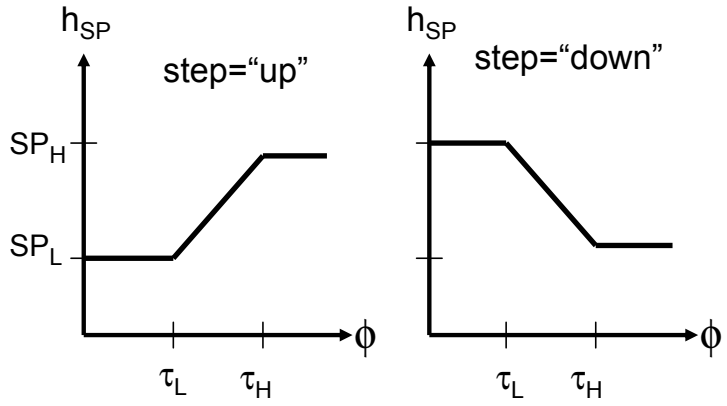

Figure 9: Binary & Linear Interpolation Set Point Control

6.1.5 SetPoint Function Control

The `trigger` token controls whether or not the setpoint controller considers the trigger values in computing the setpoint control values. If the setting is `trigger="off"` (the default), the trigger values and low and high setpoint values are ignored, the `setpoint` value is always returned. If the setting is `trigger="on"`, the response is that of the binary switch, linear interpolated section, or both, depending on the value of the `window` token.

The `window` token is used to control the setpoint response function as either a binary switch, linear interpolated section, or both. The corresponding assignments to this token are:

1. Binary Switch: `window="outside"`
2. Linear Interpolation: `window="inside"`
3. Binary & Linear Interpolation: `window="all"`

The default value is: `window="all"`.

The token `step` controls the abscissa transition behavior of the trigger functions. If `step="up"` (the default) a step-up behavior is returned. If `step="down"` a step-down behavior is returned.

The `control` attribute can be used to deactivate the controller. If the value of `control` is set to any value other than "on", the controller will be deactivated. This means that the control output will be forced to a value of 1, no control output variations will occur. Since the control outputs are applied as amplitude modulation factors to the watermover flow, the watermover flow will default to it's uncontrolled values.

6.2 SetPoint XML

The controller environments available for the SetPoint controller are shown in Table 6.2.

environment	attribute	meaning
`<setpointctrl>`		SetPoint controller definition
	cid	positive (cid>0) controller id
	label	optional controller label
	wmID	ID of watermover to be controlled
	control	'on' or 'off'
	trigger	'on' or 'off'
	window	'outside', 'inside' or 'all'
	step	'up' or 'down'
	setpoint	constant setpoint value
	setlow	low setpoint value
	sethigh	high setpoint value
	triglow	low trigger value
	trighigh	high trigger value
`<*monitor>`		State variable specification

Table 6.2. Setpoint Controller XML

The following example illustrates a SetPoint controller applied to a watermover.

```
<controller id="1">
  <setpointctrl cid="101" label="SPCtrl 1:" wmID="1"
   window="all" setpoint="495.0" setlow="0.0" sethigh="1.0"
   triglow="505.0" trighigh="506.0" trigger="on" step="down">
    <segmentmonitor id="1" attr="head"></segmentmonitor>
  </setpointctrl>
</controller>
```

A full example of the Setpoint controller applied in a RSM test case is shown in 15.5.

7 Fuzzy Controller

The RSM incorporates a generic, multi-variate, fuzzy controller through access to the Fuzzy Control Library [3]. The Fuzzy Control Library is a C++ implementation of a generic fuzzy controller based on the Fuzzy Control Language (FCL) as defined by the International Electrotechnical Commission (IEC) standard for Fuzzy Control Programming [4]. This standard defines nomenclature and methodology for the application of rule-based linguistic (fuzzy) control modules.

Fuzzy control has significant advantages over canonical control structures in cases where the system is nonlinear and/or has no closed form transfer function representation. Another inherent strength of fuzzy control is its ability to generate outputs based on inferencing of multi-inputs which are constrained by multiple, and possibly overlapping boundary conditions. Fuzzy control is essentially an expert system which relies upon a rule-base to define the variable control constraints. The user must also define input/output fuzzy membership functions which effect the inferencing and constraint application.

7.1 Fuzzy Control Implementation

The fuzzy controller requires an input FCL file which specifies the input/output variables, fuzzy membership functions, and rule-base. Refer to the FCL Standard [4] for nomenclature, descriptions and examples of FCL files. In addition to the FCL file, XML attributes specific to the fuzzy controller are also required to establish input/output variable definitions in relation to RSM watermover monitors. Valid attributes are shown in Table 7.4. The fuzzy controller is implemented through application programming interface function calls to the Fuzzy Control Library [3].

7.2 Fuzzy Control Input/Output Terms

The ability for input and output logical decision variables to cover a range of values rather than just true or false is a central advantage of fuzzy logic compared to boolean logic. Assignment of the degree of membership (μ) to an input or output data value (χ) is performed in the fuzzification or defuzzification of input or output terms. In the fuzzy controller there are five input/output terms available for fuzzification and defuzzification:

1. Ramp
2. Triangle
3. Trapezoid
4. Rectangle
5. Singleton

Figure 10 depicts each of the fuzzy terms, each term is described below.

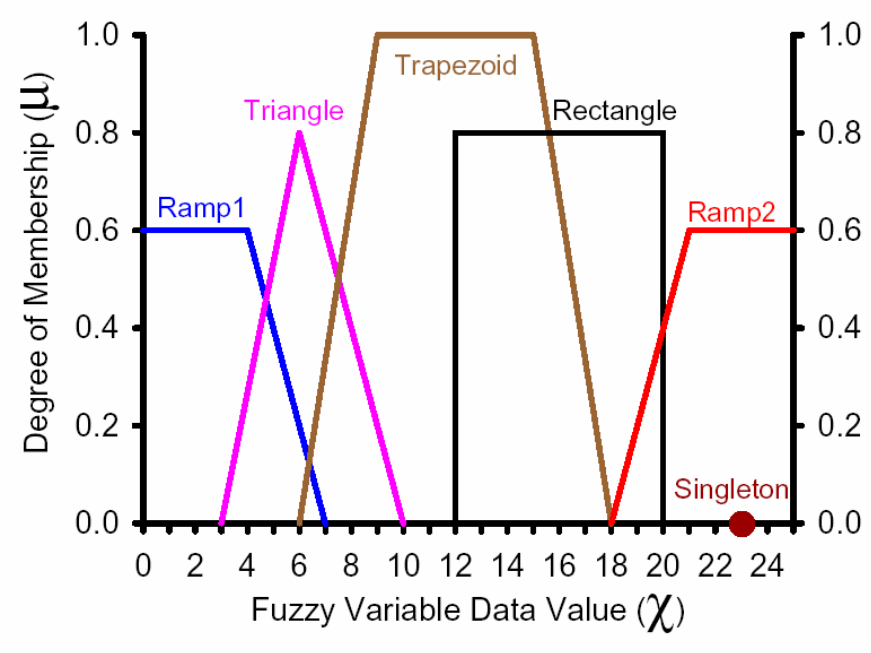

Figure 10: Available fuzzy terms

7.2.1 Ramp Term

The ramp term is used at the edges of an input/output variable domain. This term is defined in the FCL file by two (χ, μ) points. The first ramp in figure 10 would be defined as:

```
TERM ramp1 := (4, 0.6) (7, 0);
```

All data values (χ) which fall in the range $\chi < 4$, will be assigned μ=0.6. Data values which fall in the range $\chi > 7$ will not be classified as a ramp1 term. This means that if $\chi > 7$, then a fuzzy rule of the form

```
IF inputVar IS ramp1 THEN ouputVar IS low;
```

will be off (i.e. the output membership value of the `ouputVar` term `low` will be zero.) The last ramp term in figure 10 would be defined as:

```
TERM ramp2 := (18, 0) (21, 0.6);
```

All data values (χ) which fall in the range $\chi > 21$, will be assigned μ=0.6.

7.2.2 Triangle Term

The triangle term requires three (χ, μ) points. The triangle term shown in figure 10 is defined by:

```
TERM triangle := (3, 0) (6, 0.8) (10, 0);
```

All data values which fall outside of the interval $3 < \chi < 10$ will not be classified as a triangle term (i.e. the triangle term membership value will be zero for that fuzzy variable.)

7.2.3 Trapezoid Term

A trapezoid term is one of the most frequently used fuzzy input/output terms. A trapezoid requires four (χ, μ) points. To qualify as a trapezoid, the ordinates of the 1st & 2nd, and 3rd & 4th points cannot be the same value. The trapezoid term of figure 10 is defined by:

```
TERM trapezoid := (6, 0) (9, 1) (15, 1) (18, 0);
```

7.2.4 Rectangle Term

The rectangle term is typically not used in fuzzy logic, as it precludes the smooth transition of variable membership from one term to another. It can however be useful in cases where an on/off behavior is desired. The rectangle term requires four (χ, μ) points. The rectangle term shown in figure 10 is defined by:

```
TERM rectangle := (12, 0) (12, 0.8) (20, 0.8) (20, 0);
```

To qualify as a rectangle, the ordinates of the 1st & 2nd, and 3rd & 4th points must be the same value.

7.2.5 Singleton Term

The singleton is not a true fuzzy logic term. It does not classify an input term based on the range of the data, and does not provide a center-of-mass or other inference output value for output terms. Rather, it simply assigns a data value to a term. This can be useful in isolating behavioral problems of the fuzzy controller, or in cases where precise control of output variables is desired. The singleton does not require a (χ, μ) point, but just specifies the data value χ. It is therefore misleading to interpret the singleton term shown in figure 10 as being associated with an input/output data value χ. See the example of the fuzzy controller in section 15.7 for a demonstration. Notwithstanding the crisp nature of the singleton terms, the user must realize that if multiple singleton terms are active as a result of rule validation, the singleton values will be still be subjected to the fuzzy inferencing processes of aggregation, activation and accumulation, thereby resulting in a variable value that may not match any of the singleton values. A valid FCL invocation of a singleton term would be:

```
TERM singleton := 23;
```

7.3 FCL Syntax Validation

Included in the RSM distribution is a program for checking the syntax and validity of an FCL file without having to run the RSM. The program `fcl_valid` is in the `fcl_lib` directory. This program will read a FCL file as input, parse the FCL file, create an input variable map, output variable map, and rule map exactly as is done in the RSM fuzzy controller. Any FCL syntax errors, or errors in creation of the input/output/rule maps will be detected and printed to the output. The usage of the program is:

```
fcl_valid controller.fcl
```

where `controller.fcl` is the name of the fuzzy controller FCL definition file. Note that this program cannot check the input and output variable linkage between the FCL file and the XML controller definition files.

An example of `fcl_valid` is shown below. The FCL file to which `fcl_valid` is applied is:

```
// S8
//      1) IF stage in Miami Canal > 12.5 ft AND not enough
//         gravity flow through the spillway from low WCA3, pump on
FUNCTION_BLOCK S8
VAR_INPUT
        Miami_Canal_Stage : REAL;
END_VAR
VAR_OUTPUT
        S8_Pump : REAL;
END_VAR
FUZZIFY Miami_Canal_Stage
        TERM low    := (9, 1) (10, 0);
        TERM medium := (9, 0) (10, 1) (12, 1) (13, 0);
        TERM high   := (12.5, 0) (13, 1);
END_FUZZIFY
DEFUZZIFY S8_Pump
          // All outputs are singletons
          TERM off     := 0.;
          TERM qtr_on  := 0.25;
          TERM half_on := 0.5;
          TERM 3qtr_on := 0.75;
          TERM on      := 1.;
          ACCU: MAX;
          METHOD: COG;
          DEFAULT:= 0;
          RANGE:= (0, 1);
END_DEFUZZIFY
RULEBLOCK No1
          AND : MIN;
          OR  : MAX;
          ACT : MIN;
          RULE 1: IF Miami_Canal_Stage IS high
                  THEN S8_Pump IS on;
          RULE 2: IF Miami_Canal_Stage IS low OR Miami_Canal_Stage IS medium
                  THEN S8_Pump IS off;
END_RULEBLOCK
END_FUNCTION_BLOCK
```

The corresponding output from `fcl_valid` is shown below:

```
[linuxserv1] lec> fcl_valid S8.fcl
-> : fcl_valid() Start : (0)
->InputVariables Map
================================================
FuzzyInputClass: Miami_Canal_Stage varName: Miami_Canal_Stage varType: REAL
        FuzzyInputTerm: high
        varName: Miami_Canal_Stage termName: high termType: Ramp
         (12.5, 0)  (13, 1)
        FuzzyInputTerm: low
        varName: Miami_Canal_Stage termName: low termType: Ramp
         (9, 1)  (10, 0)
        FuzzyInputTerm: medium
        varName: Miami_Canal_Stage termName: medium termType: Trapezoid
         (9, 0)  (10, 1)  (12, 1)  (13, 0)
================================================
<-InputVariables Map
->OutputVariables Map
================================================
FuzzyOutputClass: S8_Pump varName: S8_Pump varType: REAL
        ACCU: MAX METHOD: COG DEFAULT: 0 RANGE: (0, 1)
        FuzzyOutputTerm: 3qtr_on
        varName: S8_Pump termName: 3qtr_on termType: Singleton
        singleton: 0.75, 0
        FuzzyOutputTerm: half_on
        varName: S8_Pump termName: half_on termType: Singleton
        singleton: 0.5, 0
        FuzzyOutputTerm: off
        varName: S8_Pump termName: off termType: Singleton
        singleton: 0, 0
        FuzzyOutputTerm: on
        varName: S8_Pump termName: on termType: Singleton
        singleton: 1, 0
        FuzzyOutputTerm: qtr_on
        varName: S8_Pump termName: qtr_on termType: Singleton
        singleton: 0.25, 0
```

```
================================================
<-OutputVariables Map
->Rules Map
================================================
FuzzyRuleClass: 1 ruleName: 1
        AND: MIN OR: MAX ACT METHOD: MIN
        Conditions:
        AND SubCondition: InputVar: Miami_Canal_Stage InputTerm: high
        Conclusions:
        Conclusion: OutputVar: S8_Pump OutputTerm: on weight: 1
FuzzyRuleClass: 2 ruleName: 2
        AND: MIN OR: MAX ACT METHOD: MIN
        Conditions:
        OR SubCondition: InputVar: Miami_Canal_Stage InputTerm: low
        OR SubCondition: InputVar: Miami_Canal_Stage InputTerm: medium
        Conclusions:
        Conclusion: OutputVar: S8_Pump OutputTerm: off weight: 1
================================================
<-Rules Map
<- : fcl_valid() Complete. : (0)
```

7.4 Fuzzy Control XML

The controller environments available for the Fuzzy controller are shown in Table 7.4.

environment	attribute	meaning
`<fuzctrl>`		Fuzzy controller definition
	cid	positive (cid>0) controller id
	label	optional controller label
	wmID	ID of watermover to be controlled
	control	'on' or 'off'
	fcl	name of FCL file
`<varIn>`		input variable(s)
	name	input variable name
	monitor	monitor specification
	monID	monitor ID
	monType	monitor type
`<varOut>`		output variable
	name	output variable name
`<target>`		Target state specification (optional)
`<*monitor>`		State variable specification

Table 7.4. Fuzzy Controller XML

The `control` attribute can be used to deactivate the controller. If the value of `control` is set to any value other than "on", the controller will be deactivated. This means that the control output will be forced to a value of 1, no control output variations will occur. Since the control outputs are applied as amplitude modulation factors to the watermover flow, the watermover flow will default to it's uncontrolled values.

The `fcl` attribute must evaluate to a valid Fuzzy Control Language (FCL) file specification. The `varIn` environment and attributes will link the HSE state values (via monitors) to the controller input variables. The `<varIn> name=` attribute must correspond to a `VAR_INPUT` defined in the specified FCL file. The `monitor`, `monID` and `monType` attributes must match that of the monitor defined in the controller section.

The `<varOut>` argument must correspond to a `VAR_OUTPUT` variable specified in the FCL file. The *monitor definition (in this case a `segmentmonitor`)

will link the RSM input state variable to the `<varIn>` argument as described above.

The following example illustrates a single input Fuzzy controller applied to a watermover.

```
<controller id="1">
  <!-- Controller for discharge from segment 1 -->
  <fuzctrl cid="101" wmID="1" fcl="hq1.fcl" label="fcl1"
    <varIn name="segment1Head" monitor="segmentmonitor"
     monID="1" monType="head"> </varIn>
    <varOut name="control1Out"> </varOut>
    <segmentmonitor id="1" attr="head"></segmentmonitor>
  </fuzctrl>
  <!-- Controller for pumping into segment 4 -->
  <fuzctrl cid="102" wmID="2" fcl="hq1.fcl" label="fcl2"
    <varIn name="segment4Head"monitor="segmentmonitor"
     monID="4" monType="head"> </varIn>
    <varOut name="control1Out"> </varOut>
    <segmentmonitor id="4" attr="head"></segmentmonitor>
  </fuzctrl>
</controller>
```

The following example illustrates a dual input Fuzzy controller applied to a watermover. The `<varIn>` arguments must correspond to `VAR_INPUT` variables defined in the specified FCL file. The `<varIn>` argument is a list of input variables, where each input variable specification also defines the RSM state variable monitor information for that input variable. This information is required to establish correct input variable state information links to the fuzzy input variables. The `monitor`, `monID` and `monType` attributes must match that of a monitor defined in the controller section.

The `varOut` argument must correspond to a `VAR_OUTPUT` variable specified in the FCL file. The *monitor definitions (in this case `segmentmonitor`) will link the RSM input state variables to the `<varIn>` arguments.

```
<controller id="1">
  <!-- Controller for discharge from segment 1 -->
  <fuzctrl cid="101" wmID="1" fcl="hq1.fcl" label="fcl1"
    <varIn name="segment1Head" monitor="segmentmonitor"
     monID="1" monType="head"> </varIn>
    <varIn name="segment2Head" monitor="segmentmonitor"
     monID="2" monType="head"> </varIn>
    <varOut name="control1Out"> </varOut>
    <segmentmonitor id="1" attr="head"></segmentmonitor>
    <segmentmonitor id="2" attr="head"></segmentmonitor>
  </fuzctrl>
  <!-- Controller for pumping into segment 4 -->
  <fuzctrl cid="102" wmID="2" fcl="hq1.fcl" label="fcl2"
    <varIn name="segment3Head" monitor="segmentmonitor"
     monID="3" monType="head"> </varIn>
    <varIn name="segment4Head" monitor="segmentmonitor"
     monID="4" monType="head"> </varIn>
    <varOut name="control1Out"></varOut>
    <segmentmonitor id="3" attr="head"></segmentmonitor>
    <segmentmonitor id="4" attr="head"></segmentmonitor>
  </fuzctrl>
</controller>
```

Examples of the fuzzy controller can be found in section 15.7.

8 User Controller

The user controller provides a facility for the user to independently develop a control algorithm applied to a watermover. The user develops a control algorithm in C/C++, then compiles the control routine(s) into a shared object library. The `<userctrl>` implements a shared-library loader and function pointer interface which calls the user-defined control function(s) at each timestep. Each `<userctrl>` will maintain it's own shared object and function pointer information, allowing the user to define multiple control functions inside a single shared object so that individual controllers may be enacted by selected functions which reside inside a single shared object. It is also possible to define separate shared objects for each controller. The user defined functions can take advantage of several RSM application programming interface (API) functions to assist in accessing input state variables and setting output values.

8.1 Implementation

Implementation of the user controller is dependent upon the creation of a shared object file that contains the control logic function for each controller. Each function must return a `double` floating point value, assumed to be in the interval [0,1], which is used to amplitude modulate the flow of the watermover. User defined controller and supervisor shared-object codes must include the file `mseIO.h`.

8.2 User Controller XML

The controller XML environments available for the User controller are shown in Table 8.2.

environment	attribute	meaning
<userctrl>		user controller definition
	cid	positive (cid>0) controller id
	label	optional controller label
	wmID	ID of watermover to be controlled
	module	name of shared library
	func	name of the function
	libType	type of shared object (C, C++)
	control	'on' or 'off'
	ctrlMin	minimum control output value
	ctrlMax	maximum control output value
<varIn>		input variable(s)
	name	input variable name
	source	"monitor", "xml"
	monitor	monitor specification
	monID	monitor ID
	monType	monitor type
<*monitor>		State variable specification

Table 8.2. User Controller XML

The `control` attribute can be used to deactivate the controller. If the value of `control` is set to any value other than "on", the controller will be deactivated. This means that the control output will be forced to a value of 1, no control output variations will occur. Since the control outputs are applied as amplitude modulation factors to the watermover flow, the watermover flow will default to it's uncontrolled values.

The `<varIn>` environments define the state input(s) and provide a link between the inputs defined in the XML file and the RSM state variables. The `name=` argument defines a key which is used in the user-defined function code to access input state variable. Each `<varIn>` entry can have one of two `source=` attributes, indicating the source of the data input: `"monitor"` or `"xml"`. The default is `source="monitor"` in which case the `varIn` will have an associated `<*monitor>` (i.e. `<segmentmonitor>`) entry. The `monitor=`, `monID=` and `monType=` attributes must match the attributes of the associated state `<*monitor>`. For example, the following are valid XML entries for a `varIn` and `monitor` pair:

```
<varIn name="Segment1"  monitor="segmentmonitor"
       monID="1" monType="head"></varIn>
<segmentmonitor id="1" attr="head"></segmentmonitor>
```

It is also possible to define 'static' input variables directly in the XML specification of the `varIn`, in this case `source="xml"`, as described in section 8.9.1. The user can also obtain input variables from objects of the MSE Network (see section 14, if one is defined, through the use of API functions (sections 8.10.8-8.10.13). Examples of this usage are presented in section 8.9.2.

The following example illustrates a User controller applied to a water-mover.

```
<controller id="1">
  <!-- Controller for discharge from segment 1 -->
  <userctrl cid="101" label="Segment 1 Ctrl " wmID="1" libType="C++"
            module="./UserCtrl.so" func="Segment1_Control" >
     <varIn name="Segment1"  monitor="segmentmonitor"
            monID="1" monType="head"></varIn>
     <varIn name="Segment4"  monitor="segmentmonitor"
            monID="4" monType="head"></varIn>
     <segmentmonitor id="1" attr="head"></segmentmonitor>
     <segmentmonitor id="4" attr="head"></segmentmonitor>
  </userctrl>
</controller>
```

In this example the shared object named UserCtrl.so (`module="./UserCtrl.so"`) is loaded from the current directory to call the control function named as `func="Segment1_Control"`. This function will receive two state input values named `Segment1` and `Segment4`. The state variables will reside in an `InputState` C++ map, with keys `Segment1` and `Segment4`, used to access pointers to `InputState` data structures, which contain the `stateIn` variable. The `stateIn` values will be assigned from the appropriate `segmentmonitor` at each time step, prior to calling the control function.

An example of a user controller applied in an RSM test case can be found in section 15.8.

8.3 User Controller Library Compilation

To convert the above C++ code into a shared object with the name UserCtrl.so, assuming that the function is in a file named userctrl.cc, the following command can be used in Linux:

```
gcc userctrl.cc -Bsymbolic -shared -o UserCtrl.so
```

8.4 User Controller Initialization and Cleanup

If the user desires to have a one-time initialization call, and one-time cleanup call made to the library at the time the library is loaded and unloaded respectively, the user must define two functions within the library:

1. `void _init() { };`
2. `void _fini() { };`

The `_init()` function will be called after the shared library loader successfully imports the shared library, and the `_fini()` function will be called before the library is unloaded. To prevent linkage conflicts with common standard library functions, add the -nostdlib argument to the command:

```
gcc userctrl.cc -Bsymbolic -shared -o UserCtrl.so -nostdlib
```

8.5 User Controller and Statically Linked HSE

The current version of RSM does not support user defined controllers developed in C++ if the RSM is statically linked. If the RSM is dynamically linked, then C++ user defined shared libraries are supported.

8.6 No Input State Variables (varIn)

In the event that a user defined controller does not have any input state variables, or if the controller is being used as an interface from a supervisor which is continually overriding the control output, then the `varIn` input should have the `name` attribute set to `"none"`. In these cases an input data monitor is not created for the controller. As an example:

```
<userctrl cid="101" wmID="1" control="on"
          module="./spvr_test_ctrl.so" func="Dummy_Control" >
   <varIn name="none"></varIn>
</userctrl>
```

8.7 User Controller Criteria

The following criteria apply to the `<userctrl>`:

1. Ensure that the location of your library is included in the environment variable `LD_LIBRARY_PATH`.
2. The function must include the verb—mseIO.h— header file.
3. The function must include the `state_mapIO.cc` file to access MSE API functions.
4. C++ shared library functions must accept a single argument, a pointer to a C++ STL container: `map<string, InputState*>`.
5. C shared library functions must accept three input arguments. First is an integer, the number of variables. Second is an array of character pointers, each array element listing a variable name. Third is an array of floating point `double` pointers, each reference is the current state of the corresponding variable (with the same array index) listed in the array of variable names.
6. The function must return a double floating point value, the control output.
7. The expected range of control output is [0, 1]. The output value is applied as a multiplier to the watermover flow value.
8. If you compile with a C++ compiler, declare the functions as `extern "C"` to avoid name mangling in the shared object.

8.8 C User Controller Interface

If the shared library is developed in C, the controller functions are called from the MSE with three input variables as shown in the prototype below:

```
double MyControl( int     varInNum,
                  char  **varInNames,
                  double *varInValues );
```

Table 8.8. C user control function prototype

The first argument is an integer which defines the number of input variables being passed to the function. This corresponds to the number of `<varIn>` variables defined in the `<userctrl>` section of the xml file. The second argument is an array of character strings. Each array element contains the name of an input variable as defined in the `<varIn="name">` section of the xml file. The third argument is an array of floating point (`double`) values, which contain the current numerical values of the state variables. The array indices of the second and third arguments match on a one-to-one correspondence. That is, for the variable with the name `varInNames[2]`, the current state value is contained in `varInValues[2]`.

The return value of the function must be a double precision floating point value. This value will be applied as the amplitude modulation factor to the watermover flow.

Use and development of C language user defined controllers and supervisors is not the recommended method for interfacing with the MSE. It is highly recommended that all user defined modules be developed in C++, which provides a more stable and robust interface than is easily possible with direct pointer access. An example of the C user defined controller interface is shown below. Note that it is the responsibility of the user to perform input variable validation, and to not execute memory access violations. These issues are handled implicitly in the C++ interface.

```
double MyControl( int     varInNum,
                  char  **varInNames,
                  double *varInValues ) {

  double controlOut = 0.;

  // Validate the input variables
  ....

  // Get input state variables
  double h1 = varInValues[0];
  double h2 = varInValues[1];

  // Provide control function based on input state variables
  ....
  return controlOut;
}
```

8.9 C++ User Controller Interface

If the user develops the shared object in C++, then the control function receives a single input variable which is a pointer to an C++ associative array (map).

```
extern "C" double MyControl( map<string, InputState*> *lpISMap );
```

Table 8.9. C++ user control function prototype

The C++ map pointed to by `lpISMap` contains pointers to `InputState` classes, one pointer for each `varIn` variable defined in the XML file. The map key to each pointer is the variable name (`varIn=""`) as defined in the `userctrl` section of the XML file. To access an input state variable the control function calls the `GetVarIn()` API function as described in section 8.10.2. The definition of the `InputState` structure can be found in the C++ source file: `mseIO.h`, which is a required header file for user defined controllers.

The return value of the function must be a double precision floating point value. This value will be applied as the amplitude modulation factor to the watermover flow. An example of a user defined control function interface is shown below.

```
#include <map>
#include "hse/src/mseIO.h"
#include "hse/mse_tools/state_mapIO.cc"

extern "C" double MyControl( map<string, InputState*> *lpISMap ) {
  double controlOut = 0.;

  // Get input state variables
  double h1 = GetVarIn( "MyControl", "Segment1", lpISMap );
  double h2 = GetVarIn( "MyControl", "Segment4", lpISMap );

  // Provide control function based on input state variables
  ....
  return controlOut;
}
```

Input state variable values are assigned at each timestep to each `varIn` variable defined in the XML file. Dynamic variables are assigned from a `<*monitor>` (`cellmonitor, segmentmonitor...`) associated with the `varIn` variable. See section 15.8 for an example. It is also possible to assign static input values directly in the input XML as described in section 8.9.1.

8.9.1 C++ User Controller XML Inputs

The C++ user controllers are able to accept static `varIn` input variables from XML entries. Three variable formats are supported: scalar, vector, matrix. The scalar is simply a single data value. A vector is a one-dimensional list of values, and the matrix is a two-dimensional table of columns and rows. To use this feature the `source` attribute of the `varIn` must be assigned `source="xml"`. Within the `varIn` environment one type of XML input variable may be created as shown below:

```
<varIn name="xmlScalar" source="xml">
   <scalar> -325.43 </scalar>
</varIn>
<varIn name="xmlVector" source="xml">
   <vector> -1.2 -3.4 -5.6 -7.8 -9.1 </vector>
</varIn>
<varIn name="xmlMatrix" source="xml">
   <matrix>  1.2  3.4 ;
             5.6  7.8 ;
             9.1  2.3 ;
   </matrix>
</varIn>
```

Vector datamembers are delimited by whitespace. Matrix columns (members of a row) are delimited by whitespace. Matrix rows are delineated by ';' or ','. All vector and matrix indices follow C/C++ semantics: 0 is the first element. To access these members from a user controller/supervisor, several interface functions are provided as described in section 8.10.

8.9.2 C++ User Controller MSE Network Inputs

An important source of synoptic state information and water control unit (WCU) operational criteria can be the MSE Network representation 14. The C++ controller is capable of accessing WCU and hydraulic structure information stored in the MSE Network objects through the use of API functions described in sections 8.10.8 - 8.10.13. An example of MSE Network inputs

that receive information from WCU named `U1` and from structure node `S1` is shown below.

```
<varIn name="U1_Name" source="mse_network"
    mse_unit="U1" unit_attr="name" >
</varIn>
<varIn name="U1_Purpose"  source="mse_network"
   mse_unit="U1" unit_attr="purpose">
</varIn>
<varIn name="S1_Name" source="mse_network"
   mse_node="S1" node_attr="name" >
</varIn>
<varIn name="S1_open" source="mse_network"
   mse_node="S1"     node_attr="open" >
</varIn>
```

8.10 C++ User Controller API functions

The MSE provides several API functions to assist the user in validation and access of input/output variables between the RSM and the user defined shared library functions. These functions can be found in the MSE source code file: `mse_tools/state_mapIO.cc`. Example usage of these functions is shown in the benchmark BM45.

8.10.1 GetMseLabel

To access the label assigned to a controller or supervisor in the XML specification, the user defined function makes a call to the `GetMseLabel()` function.

```
string GetMseLabel(string                      func,
                   map<string, InputState*> *lpInputStateMap);
```

Table 8.10.1. `GetMseLabel()` function prototype

The function returns a string value corresponding to the `label=""` assignment in the XML. The input arguments are described below.

Name	Type	Description
func	`string`	name of function in user library calling GetVarIn(), used for error reporting
lpInputStateMap	`*map<string, InputState*>`	pointer to the InputStateMap passed into the user defined function

Table 8.10.1. `GetMseLabel()` function arguments

The invocation of the `GetMseLabel()` function from the user defined controller shared library function would be:

```
extern "C" double MyControl( map<string, InputState*> *lpISMap ) {

  string controllerLabel = GetMseLabel( "MyControl", lpISMap );

  // Provide control function based on input state variable
  ....
  return controlOut;
}
```

8.10.2 GetVarIn

To access a monitor input state variable defined in the `varIn` environment of a controller or supervisor, the user defined function makes a call to the `GetVarIn()` function.

```
double GetVarIn(string                    func,
                string                    varInName,
                map<string, InputState*> *lpInputStateMap);
```

Table 8.10.2. `GetVarIn()` function prototype

The function returns a double precision floating point value corresponding to the current state value of the `varIn` input variable with the name `varInName`. The input arguments are described below.

Name	Type	Description
func	`string`	name of function in user library calling GetVarIn(), used for error reporting
varInName	`string`	name of the input variable assigned in XML `varIn`
lpInputStateMap	`*map<string, InputState*>`	pointer to the InputStateMap passed into the user defined function

Table 8.10.2. `GetVarIn()` function arguments

The first function argument `func` is a string data type that should contain the name of the function which calls the GetXML*In() function. This is used for error reporting in the event that one of the access functions fails. The second argument is a string corresponding to the `<varIn name="">` attribute of the desired input variable. The last argument is a pointer to the `inputStateMap` that was passed into the user control function by the MSE function call handler.

An example user defined controller input XML section is shown below:

```
<userctrl cid="101" label="MyControl" wmID="1" control="on"
          module="./UserCtrl.so" func="MyControl" >
   <varIn name="Segment4"  monitor="segmentmonitor"
          monID="4" monType="head">
   </varIn>
   <segmentmonitor id="4" attr="head"></segmentmonitor>
</userctrl>
```

The corresponding invocation of the `GetVarIn()` function from the user defined controller shared library function would be:

```
extern "C" double MyControl( map<string, InputState*> *lpISMap ) {

  double canalHead = GetVarIn( "MyControl", "Segment4", lpISMap );

  // Provide control function based on input state variable
  ....
  return controlOut;
}
```

8.10.3 XMLScalarValue

To access a scalar value defined in an input XML as shown in section 8.9.1, the user defined function makes a call to the `XMLScalarValue()` function.

```
double XMLScalarValue(string                    func,
                      string                    varInName,
                      map<string, InputState*> *lpInputStateMap);
```

Table 8.10.3. `XMLScalarValue()` function prototype

The function returns a double precision floating point value corresponding to the value of the `<varIn> <scalar>` input variable with the name `varInName`. The input arguments are described below.

Name	Type	Description
func	`string`	name of function in user library calling XMLScalarValue(), used for error reporting
varInName	`string`	name of the `<scalar>` input variable assigned in XML `varIn`
lpInputStateMap	`*map<string, InputState*>`	pointer to the InputStateMap passed into the user defined function

Table 8.10.3. `XMLScalarValue()` function arguments

An example user defined controller input XML section is shown below:

```
<userctrl cid="101" label="MyControl" wmID="1" control="on"
          module="./UserCtrl.so" func="MyControl" >
    <varIn name="xmlScalar" source="xml">
       <scalar> -325.43 </scalar>
    </varIn>
</userctrl>
```

The corresponding invocation of the `XMLScalarValue()` function from the user defined controller shared library function would be:

```
extern "C" double MyControl( map<string, InputState*> *lpISMap ) {

  double scalar = XMLScalarValue( "MyControl";, "xmlScalar", lpISMap );
  ....
  return controlOut;
}
```

8.10.4 XMLVector

To access the C++ STL vector container which holds input XML `<vector>`values as shown in section 8.9.1, the user defined function makes a call to the `XMLVector()` function.

```
vector <double>
   * XMLVector(string  func,
                       string                    varInName,
                       map<string, InputState*> *lpInputStateMap);
```

Table 8.10.4. `XMLVector()` function prototype

The function returns a pointer to an STL container of type `vector <double>`, a vector of double precision floating point values corresponding to the `<varIn> <vector>` input variable with the name `varInName`. The input arguments are described below.

Name	Type	Description
func	`string`	name of function in user library calling XMLVector(), used for error reporting
varInName	`string`	name of the `<vector>` input variable assigned in XML `varIn`
lpInputStateMap	`*map<string, InputState*>`	pointer to the InputStateMap passed into the user defined function

Table 8.10.4. `XMLVector()` function arguments

An example user defined controller input XML section is shown below:

```
<userctrl cid="101" label="MyControl" wmID="1" control="on"
          module="./UserCtrl.so" func="MyControl" >
    <varIn name="xmlVector" source="xml">
       <vector> -1.2 -3.4 -5.6 -7.8 -9.1 </vector>
    </varIn>
</userctrl>
```

The corresponding invocation of the `XMLVector()` function from the user defined controller shared library function would be:

```
extern "C" double MyControl( map<string, InputState*> *lpISMap ) {
  string func = "MyControl";

  vector <double> *xmlVector = XMLVector( func, "xmlVector", lpISMap );
  ....
  return controlOut;
}
```

Important: Since this function returns a pointer to an STL container, the user must exercise caution in accessing offset values referenced to the pointer. C++ provides checked access for the `vector` STL container via the `at()` member function. To access element `i` of the `xmlVector`:

```
   if (xmlVector) { value = xmlVector->at(i); }
```

8.10.5 XMLMatrix

To access the C++ STL vector container which holds input XML `<matrix>` values as shown in section 8.9.1, the user defined function makes a call to the `XMLMatrix()` function.

```
vector<vector<double>>
   *XMLMatrix(string func,
              string                      varInName,
              map<string, InputState*> *lpInputStateMap);
```

Table 8.10.5. `XMLMatrix()` function prototype

The function returns a pointer to an STL container of type `vector<vector <double>>`, a vector of vectors of double precision floating point values corresponding to the `<varIn> <matrix>` input variable with the name `varInName`. The input arguments are described below.

Name	Type	Description
func	`string`	name of function in user library calling XMLMatrix(), used for error reporting
varInName	`string`	name of the `<matrix>` input variable assigned in XML `varIn`
lpInputStateMap	`*map<string, InputState*>`	pointer to the InputStateMap passed into the user defined function

Table 8.10.5. `XMLMatrix()` function arguments

An example user defined controller input XML section is shown below:

```
<userctrl cid="101" label="MyControl" wmID="1" control="on"
          module="./UserCtrl.so" func="MyControl" >
    <varIn name="xmlMatrix" source="xml">
         <matrix>  1.2   3.4  5.6   7.8 ;
                  -1.2  -3.4 -5.6  -7.8 ;
                   9.1   2.3  4.5   6.7 ;
         </matrix>
    </varIn>
</userctrl>
```

The corresponding invocation of the `XMLMatrix()` function from the user defined controller shared library function would be:

```
extern "C" double MyControl( map<string, InputState*> *lpISMap ) {
  string func = "MyControl";

  vector< vector<double> > *xmlMatrix = XMLMatrix( func,
                                                   "xmlMatrix",
                                                   lpISMap );
  ....
  return controlOut;
}
```

Important: Since this function returns a pointer to an STL container, the user must exercise caution in accessing offset values referenced to the pointer. C++ provides checked access for the `vector` STL container via the `at()` member function. To access element `i, j` of the `xmlMatrix`:

```
   if (xmlMatrix) { value = xmlMatrix->at(i).at(j); }
```

8.10.6 XMLVectorValue

To access a single element of a `<varIn> <vector>` value defined in an input XML as shown in section 8.9.1, the user defined function makes a call to the `XMLVectorValue()` function. This is safer than accessing the STL `vector` directly by a pointer and offset, since the offset validation is done by the `XMLVectorValue()` function.

```
double XMLVectorValue(string                    func,
                      string                    varInName,
                      int                       i,
                      map<string, InputState*> *lpInputStateMap);
```

Table 8.10.6. `XMLVectorValue()` function prototype

The function returns a double precision floating point value corresponding to the value of the i^{th} element of the `<varIn> <vector>` input variable with the name `varInName`. The input arguments are described below.

Name	Type	Description
func	`string`	name of function in user library calling XMLVectorValue(), used for error reporting
varInName	`string`	name of the `<scalar>` input variable assigned in XML `varIn`
i	`int`	0 based offset of element
lpInputStateMap	`*map<string, InputState*>`	pointer to the InputStateMap passed into the user defined function

Table 8.10.6. `XMLVectorValue()` function arguments

In the `XMLVectorValue()` function, the third argument is the (zero-offset) element number of a vector element. If invalid indices are specified the function will return a value of 0 and print an error message to cerr. An example user defined controller input XML section is shown below:

```
<userctrl cid="101" label="MyControl" wmID="1" control="on"
          module="./UserCtrl.so" func="MyControl" >
    <varIn name="xmlVector" source="xml">
       <vector> -1.2 -3.4 -5.6 -7.8 -9.1 </vector>
    </varIn>
</userctrl>
```

The corresponding invocation of the `XMLVectorValue()` function from the user defined controller shared library function would be:

```
extern "C" double MyControl( map<string, InputState*> *lpISMap ) {
  string func = "MyControl";

  double vectorVal = XMLVectorValue( func, "xmlVector", 2, lpISMap );
  ....
  return controlOut;
}
```

The function call above would return the value of -5.6.

8.10.7 XMLMatrixValue

To access a single element of a `<varIn> <matrix>` value defined in an input XML as shown in section 8.9.1, the user defined function makes a call to the `XMLMatrixValue()` function. This is safer than accessing the STL `vector` directly by a pointer and offset, since the offset validation is done by the `XMLMatrixValue()` function.

```
double XMLMatrixValue(string                    func,
                      string                    varInName,
                      int                       row,
                      int                       col,
                      map<string, InputState*> *lpInputStateMap);
```

Table 8.10.7. `XMLMatrixValue()` function prototype

The function returns a double precision floating point value corresponding to the value of the element at the `row, col` of the `<varIn> <matrix>` input variable with the name `varInName`. The input arguments are described below.

Name	Type	Description
func	`string`	name of function in user library calling XMLMatrixValue(), used for error reporting
varInName	`string`	name of the `<scalar>` input variable assigned in XML `varIn`
row	`int`	0 based offset of element row
col	`int`	0 based offset of element column
lpInputStateMap	`*map<string, InputState*>`	pointer to the InputStateMap passed into the user defined function

Table 8.10.7. `XMLMatrixValue()` function arguments

In the `XMLMatrixValue()` function, the third and fourth arguments of the function are indices corresponding to the (zero-offset) row and column

of the data in the ¡matrix¿ subnodes. If invalid indices are specified the function will return a value of 0 and print an error message to cerr. An example user defined controller input XML section is shown below:

```
<userctrl cid="101" label="MyControl" wmID="1" control="on"
          module="./UserCtrl.so" func="MyControl" >
   <varIn name="xmlMatrix" source="xml">
       <matrix>  1.2   3.4  5.6   7.8 ;
                -1.2  -3.4 -5.6  -7.8 ;
                 9.1   2.3  4.5   6.7 ;
       </matrix>
   </varIn>
</userctrl>
```

The corresponding invocation of the `XMLMatrixValue()` function from the user defined controller shared library function would be:

```
extern "C" double MyControl( map<string, InputState*> *lpISMap ) {
  string func = "MyControl";

  double matrixVal = XMLMatrixValue( func, "xmlMatrix", 2, 3, lpISMap );
  ....
  return controlOut;
}
```

The function call above would return the value of 6.7.

8.10.8 GetMSEUnitVal

To access a numeric data object in the MSE Network that is associated with a <mse_unit> (water control unit), the user defined function makes a call to the GetMSEUnitVal() function.

```
double GetMSEUnitVal(string                    func,
                     string                    varInName,
                     map<string, InputState*> *lpInputStateMap);
```

Table 8.10.8. GetMSEUnitVal() function prototype

The function returns a double precision floating point value corresponding to the value of an MSE Network data object attribute of a <mse_unit>. The attribute of the <mse_unit> is specified with the unit_attr.

Name	Type	Description
func	string	name of function in user library calling GetMSEUnitVal(), used for error reporting
varInName	string	name of the varIn input variable holding the mse_unit and mse_attr
lpInputStateMap	*map<string, InputState*>	pointer to the InputStateMap passed into the user defined function

Table 8.10.8. GetMSEUnitVal() function arguments

An example of <varIn> XML entries which access mse_unit values from the MSE Network are shown below.

```
<varIn name="WCU1_Maint"   source="mse_network"
       mse_unit="WCU1" unit_attr="maintLevel" >
</varIn>
<varIn name="WCU1_Local"   source="mse_network"
       mse_unit="WCU1" unit_attr="localLevel" >
</varIn>
<varIn name="WCU1_Purpose" source="mse_network"
       mse_unit="WCU1" unit_attr="purpose">
</varIn>
```

A corresponding entry in the MSE Network XML file for this `mse_unit` could be as follows.

```
<mse_unit name="WCU1" purpose="3">
   <unit_arcs> "Reach_1" "Reach_1S" "Reach_1E" </unit_arcs>
   <maintLevel name="maint"> <const value="5.5"> </const> </maintLevel>
   <localLevel name="local"> <const value="5.2"> </const> </localLevel>
   <inlet  name="S11 to Reach1">    "S11"   </inlet>
   <outlet name="Reach1 to S11_A" > "S11_A" </outlet>
</mse_unit>
```

In this example three input variables are passed into the controller from the `mse_unit WCU1` corresponding with the MSE water control unit data objects `maintLevel`, `localLevel` and `purpose`. The user defined function calls to access these input variables are shown below.

```
extern "C" double MyControl( map<string, InputState*> *lpISMap ) {
  string func = "MyControl";

  double WCU1_Maint = GetMSEUnitVal( func, "WCU1_Maint",   lpISMap );
  double WCU1_Local = GetMSEUnitVal( func, "WCU1_Local",   lpISMap );
  double WCU1_Purp  = GetMSEUnitVal( func, "WCU1_Purpose", lpISMap );
  ....
  return controlOut;
}
```

In the above example, the value returned into `WCU1_Maint` will be 5.5, `WCU1_Local` will be 5.2, and `WCU1_Purp` will be 3.

8.10.9 GetMSEUnitString

To access a string data object in the MSE Network that is associated with a `<mse_unit>` (water control unit), the user defined function makes a call to the `GetMSEUnitString()` function.

```
string GetMSEUnitString(string                    func,
                        string                    varInName,
                        map<string, InputState*> *lpInputStateMap);
```

Table 8.10.9. `GetMSEUnitString()` function prototype

The function returns a C++ `string` type corresponding to the value of an attribute of the `<mse_unit>` that is specified in the controller `<varIn>` attribute `unit_attr`.

Name	Type	Description
func	`string`	name of function in user library calling GetMSEUnitString(), used for error reporting
varInName	`string`	name of the `varIn` input variable holding the `mse_unit` and `mse_attr`
lpInputStateMap	`*map<string, InputState*>`	pointer to the InputStateMap passed into the user defined function

Table 8.10.9. `GetMSEUnitString()` function arguments

An example of `<varIn>` XML entries which access `mse_unit` values from the MSE Network are shown below.

```
<varIn name="WCU1_Name" source="mse_network" mse_unit="WCU1" unit_attr="name" >
</varIn>
```

A corresponding entry in the MSE Network XML file for this `mse_unit` could be as follows.

```
<mse_unit name="WCU1">
   <unit_arcs> "Reach_1" "Reach_1S" "Reach_1E" </unit_arcs>
</mse_unit>
```

In this example the `varIn` will contain a string holding the name (`WCU1`). Currently, this function is of little value, since the name of the `mse_unit` must already be known, however, it is maintained for future use where additional string values may be associated with a `mse_unit`. An example usage is shown below.

```
extern "C" double MyControl( map<string, InputState*> *lpISMap ) {
  string func = "MyControl";

  string WCU1_Name = GetMSEUnitString( func, "WCU1", lpISMap );
  ....
  return controlOut;
}
```

8.10.10 GetMSEArcVal

To access a numeric data object in the MSE Network that is associated with a `<mse_arc>`, the user defined function makes a call to the `GetMSEArcVal()` function.

```
double GetMSEArcVal(string                    func,
                    string                    varInName,
                    map<string, InputState*> *lpInputStateMap);
```

Table 8.10.10. `GetMSEArcVal()` function prototype

The function returns a double precision floating point value corresponding to the value of an MSE Network data object attribute of a `<mse_arc>`. The attribute of the `<mse_arc>` is specified with the `unit_attr`.

Name	Type	Description
func	`string`	name of function in user library calling GetMSEArcVal(), used for error reporting
varInName	`string`	name of the `varIn` input variable holding the `mse_arc` and `mse_attr`
lpInputStateMap	`*map<string, InputState*>`	pointer to the InputStateMap passed into the user defined function

Table 8.10.10. `GetMSEArcVal()` function arguments

An example of `<varIn>` XML entries which access `mse_arc` values from the MSE Network are shown below.

```
<varIn name="Arc1_Flow"     source="mse_network"
   mse_arc="Arc1" arc_attr="flow" >
</varIn>
<varIn name="Arc1_Capacity" source="mse_network"
   mse_arc="Arc1" arc_attr="capacity">
</varIn>
```

A corresponding entry in the MSE Network XML file for this `mse_arc` could be as follows.

```
<mse_arc name="Arc1" capacity="1400">
   <hse_arcs> 100 101 102 103 </hse_arcs>
   <node_source> "S11"    </node_source>
   <node_sink>   "S11_A"  </node_sink>
</mse_arc>
```

In this example two input variables are passed into the controller from the `mse_arc Arc1` corresponding to the data objects `flow` and `capacity`. The user defined function calls to access these input variables are shown below.

```
extern "C" double MyControl( map<string, InputState*> *lpISMap ) {
  string func = "MyControl";

  double Arc1_Flow = GetMSEArcVal ( func, "Arc1_Flow",     lpISMap );
  double Arc1_Cap  = GetMSEArcVal ( func, "Arc1_Capacity", lpISMap );
  ....
  return controlOut;
}
```

In the above example, the value returned into `Arc1_Capacity` will be 1400.

8.10.11 GetMSEArcString

To access a string data object in the MSE Network that is associated with a `<mse_arc>` (water control unit), the user defined function makes a call to the `GetMSEArcString()` function.

```
string GetMSEArcString(string                    func,
                       string                    varInName,
                       map<string, InputState*> *lpInputStateMap);
```

Table 8.10.11. `GetMSEArcString()` function prototype

The function returns a C++ `string` type corresponding to the value of an attribute of the `<mse_arc>` that is specified in the controller `<varIn>` attribute `unit_attr`.

Name	Type	Description
func	`string`	name of function in user library calling GetMSEArcString(), used for error reporting
varInName	`string`	name of the `varIn` input variable holding the `mse_arc` and `mse_attr`
lpInputStateMap	`*map<string, InputState*>`	pointer to the InputStateMap passed into the user defined function

Table 8.10.11. `GetMSEArcString()` function arguments

An example of `<varIn>` XML entries which access `mse_arc` values from the MSE Network are shown below.

```
<varIn name="Arc1_Name" source="mse_network" mse_arc="Arc1" arc_attr="name" >
</varIn>
```

A corresponding entry in the MSE Network XML file for this `mse_arc` could be as follows.

```
<mse_arc name="Arc1" capacity="1400">
    <hse_arcs> 100 101 102 103 </hse_arcs>
    <node_source> "S11"    </node_source>
    <node_sink>   "S11_A"  </node_sink>
</mse_arc>
```

In this example the `varIn` will contain a string holding the name (`Arc1`). Currently, this function is of little value, since the name of the `mse_arc` must already be known, however, it is maintained for future use where additional string values may be associated with a `mse_arc`. An example usage is shown below.

```
extern "C" double MyControl( map<string, InputState*> *lpISMap ) {
  string func = "MyControl";

  string Arc1_Name = GetMSEArcString( func, "Arc1", lpISMap );
  ....
  return controlOut;
}
```

8.10.12 GetMSENodeVal

To access a numeric data object in the MSE Network that is associated with a `<mse_node>`, the user defined function makes a call to the `GetMSENodeVal()` function.

```
double GetMSENodeVal(string                    func,
                     string                    varInName,
                     map<string, InputState*> *lpInputStateMap);
```

Table 8.10.12. `GetMSENodeVal()` function prototype

The function returns a double precision floating point value corresponding to the value of an MSE Network data object attribute of a `<mse_node>`. The attribute of the `<mse_node>` is specified with the `unit_attr`.

Name	Type	Description
func	`string`	name of function in user library calling GetMSENodeVal(), used for error reporting
varInName	`string`	name of the `varIn` input variable holding the `mse_node` and `mse_attr`
lpInputStateMap	`*map<string, InputState*>`	pointer to the InputStateMap passed into the user defined function

Table 8.10.12. `GetMSENodeVal()` function arguments

An example of `<varIn>` XML entries which access `mse_node` values from the MSE Network are shown below.

```
<varIn name="S1_open" source="mse_network"
       mse_node="S1" node_attr="open">
</varIn>
<varIn name="S1_close" source="mse_network"
       mse_node="S1" node_attr="close">
</varIn>
<varIn name="S1_capacity" source="mse_network"
       mse_node="S1" node_attr="capacity">
</varIn>
```

A corresponding entry in the MSE Network XML file for this `mse_node` could be as follows.

```
<mse_node name="S1" priority="1" purpose="WaterSupply" label="RatedStruct"
          designCap="3000." structure="yes" managed="yes"
          watermover="S1">
   <supply name="S1 Supply"> <const value="100"> </const> </supply>
   <open   name="S1 Open">   <rc id="2"></rc> </open>
   <close  name="S1 Close">  <const value="5.5"> </const> </close>
</mse_node>
```

In this example three input variables are passed into the controller from the `mse_node S1` corresponding to the data objects `open`, `close` and `capacity`. The user defined function calls to access these input variables are shown below.

```
extern "C" double MyControl( map<string, InputState*> *lpISMap ) {
  string func = "MyControl";

  double S1_open     = GetMSENodeVal ( func, "S1_open",     lpISMap );
  double S1_close    = GetMSENodeVal ( func, "S1_close",    lpISMap );
  double S1_capacity = GetMSENodeVal ( func, "S1_capacity", lpISMap );
  ....
  return controlOut;
}
```

In the above example, the value returned into `S1_close` will be 5.5, the other values will depend on the current state values assigned to the `mse_node` object in the MSE Network.

8.10.13 GetMSENodeString

To access a string data object in the MSE Network that is associated with a `<mse_node>` (water control unit), the user defined function makes a call to the `GetMSENodeString()` function.

```
string GetMSENodeString(string                    func,
                        string                    varInName,
                        map<string, InputState*> *lpInputStateMap);
```

Table 8.10.13. `GetMSENodeString()` function prototype

The function returns a C++ `string` type corresponding to the value of an attribute of the `<mse_node>` that is specified in the controller `<varIn>` attribute `unit_attr`.

Name	Type	Description
func	`string`	name of function in user library calling GetMSENodeString(), used for error reporting
varInName	`string`	name of the `varIn` input variable holding the `mse_node` and `mse_attr`
lpInputStateMap	`*map<string, InputState*>`	pointer to the InputStateMap passed into the user defined function

Table 8.10.13. `GetMSENodeString()` function arguments

An example of `<varIn>` XML entries which access `mse_node` values from the MSE Network are shown below.

```
<varIn name="S1_WM"  source="mse_network" mse_node="S1" node_attr="watermover" >
</varIn>
```

A corresponding entry in the MSE Network XML file for this `mse_node` could be as follows.

```
<mse_node name="S1" watermover="S1">
   <open   name="S1 Open">   <rc id="2"></rc> </open>
   <close  name="S1 Close">  <const value="5.5"> </const> </close>
</mse_node>
```

The user defined function call to access this input variable is shown below.

```
extern "C" double MyControl( map<string, InputState*> *lpISMap ) {
  string func = "MyControl";

  string S1_Watermover = GetMSENodeString( func, "S1", lpISMap );
  ....
  return controlOut;
}
```

In this example the `S1_Watermover` will contain a string holding the name (`S1`), which was assigned to the `mse_node` attribute `watermover`.

8.11 C++ User Supervisor Interface

Information on the capabilities and usage of MSE supervisors is contained in the Supervisor's manual [2]. In order to keep information on the development and usage of user defined modules and API functions within one resource, the user defined supervisory interface is described here.

User defined supervisors must be developed in C++, the supervisor function receives two input variables which are pointers to an `inputStateMap`, and an `outputControlMap` associative array.

```
extern "C" int MySupervise( map<string, InputState*>    *lpISMap,
                            map<string, OutputControl*> *lpOCMap );
```

Table 8.11. C++ user supervisor function prototype

The C++ map pointed to by `lpISMap` contains pointers to `InputState` classes, one pointer for each `varIn` variable defined in the supervisor XML file. The map key to each pointer is the variable name (`varIn=""`) as defined in the `userspvr` section of the XML file. To access an input state variable the supervisor function calls the `GetVarIn()` API function as described in section 8.10.2. The definition of the `InputState` structure can be found in the C++ source file: `mseIO.h`, which is a required header file.

In an analogous fashion, the C++ map pointed to by `lpOCMap` contains pointers to `OutputControl` classes, one pointer for each `varOut` variable defined in the supervisor XML file. The map key to each pointer is the variable name (`varOut=""`) as defined in the `userspvr` section of the XML file. To assign an output value the supervisor function calls the `SetVarOut()` API function as described in section 8.12.1. The definition of the `OutputControl` structure can be found in the C++ source file: `mseIO.h`.

The C++ supervisors must return an integer value. A return value of 0 indicates no error occurred in the supervisor function. A non-zero return value indicated that an error occurred in the user supervisor function. In this case an `MseError` exception is thrown by the MSE, ending the simulation.

An example of a user defined supervisor function interface is shown below.

```
#include <map>
#include "hse/src/mseIO.h"
#include "hse/mse_tools/state_mapIO.cc"

extern "C" int MySupervise( map<string, InputState*>    *lpISMap,
                            map<string, OutputControl*> *lpOCMap ) {

  string func = "MySupervise";
  double spvrOut1 = 0.;
  double spvrOut2 = 0.;

  double h1 = GetVarIn( func, "segment1Head", lpISMap );
  double h2 = GetVarIn( func, "segment4Head", lpISMap );

  // Provide supervsory function based on input state variable
  ....

  // Set the output variables
  if ( not SetVarOut( func, "ctrl_101", spvrOut1, lpOCMap ) ) {
    return -1;
  }
  if ( not SetVarOut( func, "ctrl_102", spvrOut2, lpOCMap ) ) {
    return -1;
  }

  return 0;
}
```

8.12 C++ User Supervisor API functions

Information on the user defined supervisory function API is included in this section in order to maintain a single API resource.

8.12.1 SetVarOut

To set the value of an output variable from a user defined supervisor, the supervisory function makes a call to the `SetVarOut` function. The `SetVarOut` function is not used with MSE controllers, as the controllers are MISO (multi input single output) processors. `SetVarOut` is used only with MSE supervisors to individually set one of the multiple output variables.

```
int SetVarOut(string                          func,
              string                          varOutName,
              double                          controlOut,
              map<string, OutputControl*> *lpOutputControlMap );
```

Table 8.12.1. `SetVarOut()` function prototype

The function returns an integer value of either 0 (failure) or 1 (success). The input arguments are described below.

Name	Type	Description
func	`string`	name of function in user library calling GetMSENodeVal(), used for error reporting
varOutName	`string`	name of the `varOut` output variable
controlOut	`double`	numeric output value of supervisor to `varOut`
lpOutputControlMap	`*map<string, OutputControl*>`	pointer to the OutputControlMap passed into the user defined supervisor function

Table 8.12.1. `SetVarOut()` function arguments

Semantics and usage of the `<varOut>` XML entries are described in the MSE Supervisors manual [2], section *Output Variables*. An example of `<varOut>` XML entries from a supervisor are shown below.

```
<!-- This uses wmID to specify a controller via cid output -->
<varOut wmID="1" func="controller" name="wm1_ctrl"> </varOut>

<!-- This uses ctrlID to set specific controller attributes -->
<varOut ctrlID="102" func="triglow" name="ctrl_102"> </varOut>
```

A corresponding C++ supervisory function could be written as shown below.

```
extern "C" int MySupervise( map<string, InputState*>    *lpISMap,
                            map<string, OutputControl*> *lpOCMap ) {

  string func = "MySupervise";
  double spvrOut1 = 0.;
  double spvrOut2 = 0.;

  // Get input state variables
  double h1 = GetVarIn( func, "segment1Head", lpISMap );
  double h2 = GetVarIn( func, "segment4Head", lpISMap );

  // Provide supervsory function based on input state variable
  ....

  // Set the output variables
  if ( not SetVarOut( func, "wm1_ctrl", spvrOut1, lpOCMap ) ) {
    return -1;
  }
  if ( not SetVarOut( func, "ctrl_102", spvrOut2, lpOCMap ) ) {
    return -1;
  }

  return 0;
}
```

In the above XML section and supervisory function, the output of `varOut` with `name="ctrl_102"` is used to dynamically assign a controller to the watermover with ID `wmID="1"`. The value of the variable `spvrOut1` in the function `MySupervise` will be converted to an integer, the controller which has an ID matching the integer value of `spvrOut1` will be set as the active controller for the watermover with ID `wmID="1"`. The second output, `spvrOut2` will be used to set the `"triglow"` parameter of the controller with the controller ID `ctrlID="102"`.

9 LP Controller

The LP controller (<lpctrl>) is not really a controller. Rather, it provides an interface to control of a watermover by the GNU Linear Programming Kit (GLPK) MSE supervisor (<glpk_supervise>). It contains no control algorithm and processes no state information. An example of a valid <lpctrl> XML entry would be:

```
<lpctrl cid="107" wmID="7" label="LPCtrl 36"> </lpctrl>
```

It is required that a GLPK supervisor is controlling the <lpctrl> controller. The supervisor must specify a (<varOut>) output variable which links the controller to a GLPK model variable. The varOut XML environment may have one of two func attributes: ControlOut or TargetFlow. Valid examples of glpk_supervise varOut environments for the lpctrl shown above are:

```
<varOut ctrlID="107" func="ControlOut" name="Gain_36"> </varOut>
```

or

```
<varOut ctrlID="107" func="TargetFlow" name="TF_36"> </varOut>
```

Every timestep for which the supervisor is executed, it will update the specified varOut attribute of the controller. If the varOut attribute is specified as ControlOut, the supervisor will directly set the controlOut variable of the controller. The value of controlOut is assumed to be in the range [0, 1], and will directly amplitude modulate the watermover flow value by this fraction.

Note that the supervisor will in general not run every timestep, while the controller will. This means that once the LP supervisor sets a controlOut value for a controller, that value will remain in effect for the controller until the next invocation of the supervisor.

If the varOut attribute is specified as TargetFlow, the supervisor is expected to return in it's output variable (name="TF_36") a target flow value in CFS for the structure. The controller will then query the watermover for the maximum available flow that is attainable under the current hydrological state conditions. Based on the requested target flow, and the maximum available flow, the controller will then compute a controlOut value in the range of [0, 1]. The resultant controlOut value will be applied to the

watermover flow.

Note that the `TargetFlow` controller will in general compute a different `controlOut` value for each timestep, since it is based on the current hydrological state, as well as the supervisor specified target flow.

See the MSE Supervisor documentation [2] for a description and examples of the `<lpctrl>` controller.

9.1 LP Controller XML

The controller environments available for the LP controller are shown in Table 9.1.

environment	attribute	meaning
`<lpctrl>`		LP controller definition
	cid	positive (cid¿0) controller id
	label	optional controller label
	wmID	ID of watermover to be controlled
	control	'on' or 'off'
	ctrlMin	minimum control output value
	ctrlMax	maximum control output value

Table 9.1. LP Controller XML

The `control` attribute can be used to deactivate the controller. If the value of `control` is set to any value other than "on", the controller will be deactivated. This means that the control output will be forced to a value of 1, no control output variations will occur. Since the control outputs are applied as amplitude modulation factors to the watermover flow, the watermover flow will default to it's uncontrolled values.

The following example illustrates a LP controller applied to a watermover.

```
<controller id="1">
  <!-- Controller for R1  -->
  <lpctrl cid="101" label="FlowCtrl R1" wmID="1"></lpctrl>
</controller>
```

The `<lpctrl>` controller will be supervised as shown in the following RSM XML excerpt:

```
<management id="1" label="glpk_supervise">
  <glpk_supervise id="801" label="glpk_supervise"
   modelFile="glpk_mse.mod" >
     <!-- Controllers to be controlled -->
     <ctrlID> 101 </ctrlID>
     <!-- Output variables to controllers -->
     <varOut ctrlID="101" func="ControlOut" name="Gain_R1"> </varOut>
     <!-- Input variables to glpk mse from hse -->
     <varIn param="PCR1" name="PCR1" monitor="ctrlmonitor"
      monID="1" monType="maxflow"> </varIn>
     <!-- Monitors from hse to VarIn variables -->
     <ctrlmonitor wmID="1" attr="maxflow"></ctrlmonitor>
  </glpk_supervise>
</management>
```

In this example, the GLPK supervisor is setting the control value of the `lpflow` controller (`ControlOut`) with the GLPK model variable named `Gain_R1`.

10 Pre-Simulation Controller Conditioning

Most modern control function implementations employ closed-loop feedback based on error minimization, as well as dynamic gain and threshold adjustments in response to changing state-variable conditions. As a result, the initial conditions of the error integrals, error derivatives, and control gains are important in the initial performance of the controllers. This requires that a history of state-variable input, and controller response be available for assessment of the error integrals and controller responses. When historical data is available, it can be applied in a preconditioning of the controller state functions. In the case of numerical modeling, it is likely that the historical data is incorporated into the model run itself, or there may not be any historical data. In such cases, there are two obvious choices available:

1. Run the model for a predetermined number of iterations to "ramp-up" the controllers, ignore the data outputs generated during these iterations, then continue to run the model as usual.

2. Run the model for predetermined number of iterations, then reset the model parameters (excepting the controllers) and restart the model run.

The first approach essentially throws away the model data generated during the controller configuration phase. The latter employs a predefined number of iterations to configure the controllers, then starts the simulation from timestep zero with the updated controller parameters.

The current controller implementation enacts the second approach. This is achieved through the use of the `preRunType` or `preRunIterations` XML attributes in the main RSM `<control>` section of the input specification file.

The `preRunIterations` token can be set to an integer number of iterations that the model will execute in a pre-run simulation mode. If this token is assigned a nonzero value, the model is run for the number of timesteps specified, during which time the controllers accumulate state-feedback information. Once the specified iterations are exhausted, the RSM clock is reset to the start of the simulation, and all hydrologic matrix values are reset. The RSM then enters its normal run mode for model execution.

The `preRunType` token can be assigned one of three values:

1. `controller`
2. `filter`
3. `all`

Currently, the `filter` and `all` values are reserved for future use and have no effect. If the `controller` value is set, the RSM will inspect each controller that is currently defined in the model run, and query the controller for its value of the number of points required for controller integration. For example, the PID controller has the parameter `nvals` for this purpose. The maximum value obtained from the controller queries is then used as the number of pre-run simulation iterations to perform.

Note that the `preRunIterations` token has precedence over the `preRunType` settings. The following example illustrates usage of the `preRunIterations` control token to enact a pre-run simulation of 110 iterations.

```
<control
  tslen="15"
  tstype="minute"
  startdate="01jan1994"
  starttime="1200"
  enddate="05jan1994"
  endtime="0600"
  alpha="0.500"
  solver="PETSC"
  method="gmres"
  precond="ilu"
  preRunIterations="110">
</control>
```

11 Global Controller On/Off

Each controller can be individually activated or deactivated with the `control` XML attribute in the controller definition section. In cases where multiple controllers are implemented in a simulation, it may be advantageous to deactivate all controllers with a single variable. This is the function of the `controllers` variable in the `<control>` section of the XML input file. The default value is `controllers="on"`. If the value is set to any value other than `"on"`, all controllers will be deactivated. This means that all control outputs will be forced to one, no control output variations will occur. Since the control outputs are applied as amplitude modulation factors to the watermover flow, the flow of all controlled watermovers will default to their uncontrolled values. An example of this usage is shown below.

```
<control
  tslen="15"
  tstype="minute"
  startdate="01jan1994"
  starttime="1200"
  enddate="05jan1994"
  endtime="0600"
  alpha="0.500"
  solver="PETSC"
  method="gmres"
  precond="ilu"
  controllers="off">
</control>
```

12 Controller Monitors

The RSM controllers are configured to allow monitoring of state variables, controller error values, controller output values, or the maximum (uncontrolled) flow of a watermover through use of the `<ctrlmonitor>` environment. The control monitors are typically defined in the `<output>`, `<management>`, or `<controller>` sections of the XML file to either record control values to an output, or provide input to supervisors or controllers. Each control monitor must define one of four attributes: `state`, `error`, `control` or `maxflow`. It also required that the controller id (`cID`) is specified to select the desired controller.

The `<ctrlmonitor>` environments available are shown in Table 12.

environment	attribute	meaning
`<ctrlmonitor>`		
	cID	ID of controller to be monitored
	attr	attribute to be monitored
	montype	type of variable: `scalar`, `vector`
	var	variable name for type `vector`

Table 12. Control Monitor XML

The `cID` attribute specifies the controller id number of the controller to be monitored. The `cID` must be a positive integer. In the case where multiple controllers are attached to a watermover, it is the mechanism that specifies which specific controller is monitored. If there is a need to monitor the effective control applied to a watermover when multiple controllers are attached to a watermover, one may use the `<wmmonitor>` as described in the following section.

If the monitor is applied to a controller which accepts multi-inputs, such as the fuzzy controller or user controller, the `var` is assigned the name of the input variable that is to be monitored and the montype is set to `vector`. (See section 15.7 or 15.8).

An example of a controller monitor is shown below for a single input, single (not multiple) controller attached to a watermover:

```
<output>
  <ctrlmonitor cID="101" attr="state">
    <dss file="pidctrl_gweir.dss"
     pn="/hse/wm1/state//15min/calc1/">
    </dss>
  </ctrlmonitor>
  <ctrlmonitor cID="101" attr="error">
    <dss file="pidctrl_gweir.dss"
     pn="/hse/wm1/error//15min/calc1/">
    </dss>
  </ctrlmonitor>
  <ctrlmonitor cID="101" attr="control">
    <dss file="pidctrl_gweir.dss"
     pn="/hse/wm1/control//15min/calc1/">
    </dss>
  </ctrlmonitor>
</output>
```

An example of a controller monitor is shown below for a multiple input controller. The first two control monitors are monitoring the multi-state input variables named `segment1Head` and `segment2Head` from watermover `wmID="1"`, one monitor for each variable.

```
<output>
  <ctrlmonitor cID="101" attr="state" var="segment1Head" montype="vector" >
    <dss file="fuzctrl_hq.dss" pn="/hse/wm1/state1//15min/calc1/">
    </dss>
  </ctrlmonitor>
  <ctrlmonitor cID="101" attr="state" var="segment2Head" montype="vector">
    <dss file="fuzctrl_hq.dss" pn="/hse/wm1/state2//15min/calc1/">
    </dss>
  </ctrlmonitor>
  <ctrlmonitor cID="101" attr="error">
    <dss file="fuzctrl_hq.dss" pn="/hse/wm1/error//15min/calc1/">
    </dss>
  </ctrlmonitor>
  <ctrlmonitor cID="101" attr="control">
    <dss file="fuzctrl_hq.dss" pn="/hse/wm1/control//15min/calc1/">
    </dss>
  </ctrlmonitor>
</output>
```

12.1 Watermover Control & Maxflow Monitors

As an alternative to using the `<ctrlmonitor>` to monitor the output control values, one may also use the `<wmmonitor>` with the attribute `attr="control"` specified. This may be more convenient than using a `<ctrlmonitor>` when the desire is to monitor the "effective" control signal applied to a watermover. This usage monitors the actual instantaneous control value applied to the watermover, even if multiple controllers are attached to a watermover.

The `<wmmonitor>` environments available are shown in Table 12.1.

environment	attribute	meaning
`<wmmonitor>`		
	wmID	Required ID of watermover to be monitored
	attr	Required attribute to be monitored
	label	Optional label

Table 12.1. Watermover Monitor XML

The possible attributes for a `<wmmonitor>` are `"flow"`, `"volume"`, `"control"`, `"maxflow"`. If `attr="control"` then the applied control value is monitored. If `attr="maxflow"` then the maximum available flow (the flow that will result if the control value = 1) is reported. An example of a `<wmmonitor>` reporting the control is shown below.

```
<wmmonitor wmID="2" attr="control">
```

13 Data Monitor Filters

The MSE relies upon assessor and filters to provide specialized and generalized data preprocessing. An assessor is a information processor intended to provide specialized aggregation or differentiation of state variables particular to a managerial decision process. Filters are generic information processors implemented to perform simple, often redundant data filtering operations. For example, a filter may apply a scalar or timeseries amplitude modulation consisting of the usual arithmetic operations (multiplication, division, addition, subtraction) or may compute simple timeseries or spatial variable statistics such as arithmetic, geometric, or other expectations, or may act as an accumulator.

The RSM implements a unified design approach for monitors, filters, and assessors based on object oriented design principles. As a result, the interfacing of these constructs from the user's perspective is particularly simple, and powerful. Assessor and filters operate in a piped FIFO fashion, as exemplified by the XML fragments below and in figure 11.

```
<WcuAssessor asmtID="101" name="Reach1" mode="wsneeds">
   <target> <dss file="Reach1Target.dss"/> </target>
</WcuAssessor>

<filter type="offset">
   <offset><dss file="Reach1Offset.dss"/></offset>
   <filter type="MovingAvg" numAvg="15">
      <assessormonitor id="101" attr="flow"></assessormonitor>
   </filter>
</filter>
```

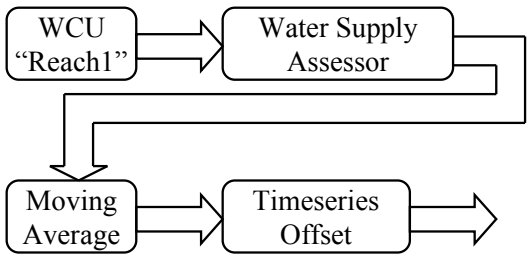

Figure 11: Filter preprocessors can be applied as piped operations.

The first XML section defines a water control unit assessor (WcuAsses-

sor) attached to the canal unit Reach1. The assessor is in water supply needs mode, which computes the flow required in the control unit to satisfy the target levels specified in the timeseries file Reach1Target.dss. The second section defines a dual-stage filter applied to the assessed flow values. An assessormonitor is used to reference the assessed flow, and serves as input to a moving average filter. The output of the moving average filter is input to an offset filter, with offset values specified by the timeseries Reach1Offset.dss. To change the data source, order, or type of operations, one simply reconfigures the XML specification. This procedure can be automated with the use of a graphical user interface software application.

There are four types of filters available:

1. Amplitude Multiplication (scale)
2. Amplitude Add/Subtract (offset)
3. Moving Average
4. Accumulation (sum)

To select a filter the user specifies the `type=""` attribute of the `<filter>` environment with one of the following values: `"mult""`, `"offset"`,`"movingavg"`,`"sum"`. If no `type=""` is specified, the filter defaults to a multiplication. The other attributes and their values are shown in Table 13.

environment	attribute	meaning
`<filter>`		
	label	optional filter label
	type	”mult”, ”offset”, ”movingavg”, ”sum”
	mult	value of multiplier
	offset	value of offset
	numAvg	number of timesteps to average over
	numSum	number of timesteps to accumulate
	order	reserved for future use

Table 13. Filter XML

14 MSE Network

The MSE network is an abstraction of the canal network and control structures suited to the needs of water resource routing and decisions. It is based on a standard graph theory representation of a flow network comprised of arcs and nodes [6, 7]. The MSE network data objects serve as state and process information repositories for management processes. They maintain assessed and filtered state information, parameter storage relevant to canals or hydraulic structure managerial constraints and variables, and serve as an integrated data source for any MSE algorithm seeking current state information.

The primary stream object in the MSE network is the Water control unit (WCU). A WCU maps a collection of HSE stream segments that are operationally managed as a discrete entity to a single arc in the MSE network. WCU's are typically bounded by hydraulic control structures, which are represented as nodes in the MSE network. Each WCU includes associative references to all inlet and outlet hydraulic flow nodes. Some of the variables stored in a structure (node) object include:

1. current flow capacity
2. maximum design flow capacity
3. reference to hydraulic watermover
4. reference to structure controller
5. operational policy water levels
6. supply
7. demand

while the WCU (arc) objects incorporate:

1. flow capacity
2. seasonal maintenance levels
3. inlet flow
4. outlet flow
5. water depth
6. water volume

Each WCU in the MSE network is referenced by a unique label, and has an associative data storage object which dynamically allocates storage for assessment results. This allows multiple, independent assessments of the WCU state. For example, one assessment of WCU inlet structure flows

might come from a graph algorithm, while another could be stored from a LP model.

This abstraction from hydrological objects to managerial objects condenses the network representation facilitating the organization and storage of relevant assessed state and process information. As an example, figure 12 depicts an HSE stream network consisting of 63 nodes and 62 segments. Some of the nodes correspond to locations of hydraulic control structures, though the association is not apparent from examination of the HSE network. Each HSE stream segment has a unique identifier which allows the modeler or MSE processor to monitor state information of the segment. However, it may be appropriate to make water management decisions based on some assessed or filtered version of aggregated HSE stream segment states.

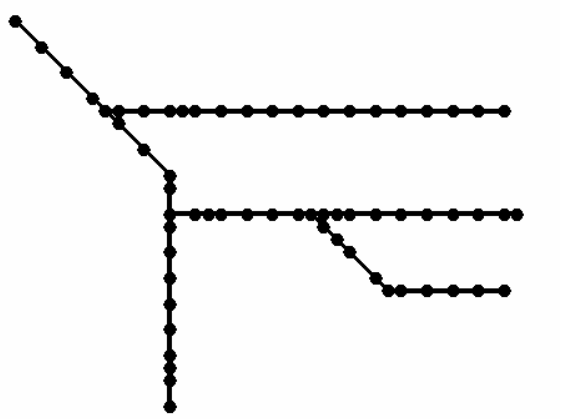

Figure 12: Example HSE stream network segments and nodes.

Consider now an abstraction of the HSE network into 10 WCU's, regulated by 11 hydraulic structures. An schematic of such a MSE network is presented in figure 13. In the MSE network each node represents a hydraulic structure which regulates a WCU, while a single line segment, or group of line segments between structures defines a WCU.

The MSE Network serves as an integrated data store for MSE algorithms which need access to aggregated, differentiated, or averaged values of hydrological variables. These processed values are typically produced by Assessors, which store the results in the proper data container of the MSE Network.

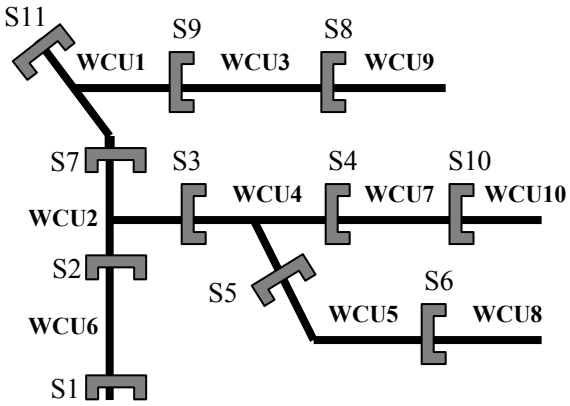

Figure 13: Example MSE network abstraction of HSE network into WCU's and structures.

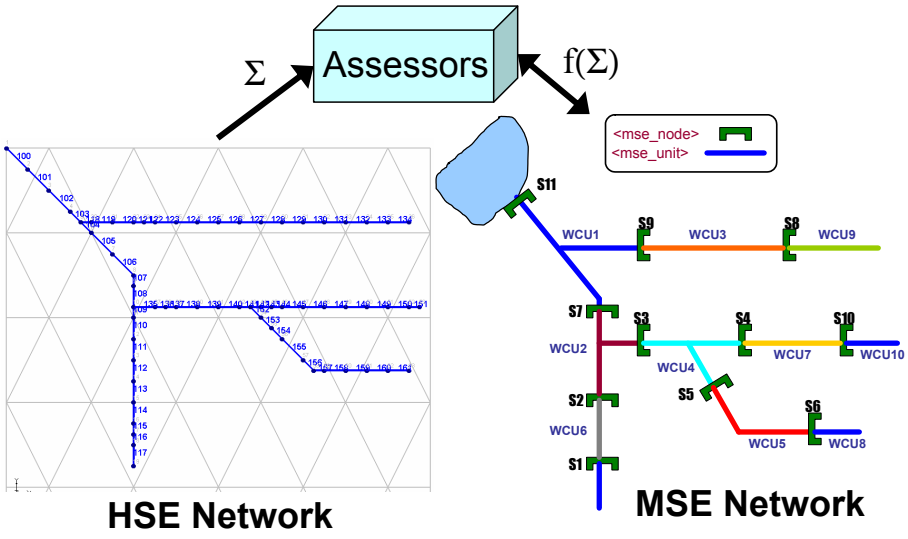

Figure 14: HSE & MSE Network

14.1 Dummy Nodes & Arcs

Generally, an `<mse_node>` has a direct correspondence to a watermover that represents a flow control structure, however, in some cases it is necessary to create a dummy `<mse_node>` to preserve the graph representation of the MSE Network. These instances are:

1. Branch in the canal network
2. Canal side-flow structure
3. Network flow endpoints

The first case is a result of the fact that an `<mse_arc>` must have only one source node, and one sink node. Therefore, where a branch in the canal network occurs, you must provide a dummy `<mse_node>` to serve as an endpoint for the branching `<mse_arc>`. An example XML of a dummy `<mse_node>` is shown in section 14.3.

The second instance refers to situations where a flow structure watermover is not an endpoint in a canal, but a 'side-flow' structure which has one bank of the canal as it's flow source. In this case it is appropriate to add a dummy `<mse_node>` at this point, which will be attached to a dummy `<mse_arc>`, which in turn attaches to the flow structure `<mse_node>`.

In the third case, if one wishes to implement one of the graph-flow algorithms as an assessor or supervisor, it is required that the flow network have only one inlet (source) node and one outlet (sink) node. Again, a dummy `<mse_node>` can be created for each of these, with one dummy `<mse_arc>` added between the dummy source/sink node and each structure watermover source/sink node in the modeled flow network.

14.2 MSE Network XML

As with other RSM model inputs, the WCU mapping from the HSE stream network to the MSE Network, as well as assignment of canal and structure parameters is performed with an input XML entry. The XML Document Type Definition (DTD) file for the MSE Network is distinct from the HSE DTD file. The MSE Network DTD file is: `mse_network.dtd`, and must be appropriately referenced in the MSE Network XML file header, as shown below.

```
<?xml version="1.0"?>
<!DOCTYPE mse_network SYSTEM "../mse_network.dtd" [
]>
```

The excerpt below shows basic elements in the construction of an MSE network.

```
<mse_network name="Test Network">
  <mse_arcs>
    <mse_arc name="Reach_1" capacity="1400">
      <hse_arcs> 100 101 102 103 </hse_arcs>
      <node_source> "S11"    </node_source>
      <node_sink>   "S11_A"  </node_sink>
    </mse_arc>
    <!-- more mse_arc entries.... -->
  </mse_arcs>
  <mse_nodes>
    <flow_source> "S1"   </flow_source>
    <flow_sink>   "Sink" </flow_sink>
    <mse_node name="S11" purpose="WaterSupply" designCap="3000.">
      <supply name="S11 Supply"> <const value="100"> </const> </supply>
      <open   name="S11 Open">   <rc id="2"></rc> </open>
      <close  name="S11 Close">  <const value="5.5"> </const> </close>
    </mse_node>
    <!-- more mse_node entries.... -->
  </mse_nodes>
  <mse_units>
    <mse_unit name="WCU1">
      <unit_arcs> "Reach_1" "Reach_1S" "Reach_1E" </unit_arcs>
      <maintLevel name="maint"> <const value="5.5"> </const> </maintLevel>
      <inlet  name="S11 inlet">  "S11" </inlet>
      <outlet name="S7 outlet" > "S7"  </outlet>
      <outlet name="S9 outlet" > "S9"  </outlet>
    </mse_unit>
    <!-- more mse_unit entries.... -->
  </mse_units>
</mse_network>
```

The above example does not define an operational MSE Network, but simply illustrates the structure and attributes of the XML file. Working examples can be found in the Benchmarks BM63, BM49, BM48. The following sections detail the contents of the MSE Network XML environments.

14.2.1 MSE Arc

The `<mse_arcs>` environment can only contain `<mse_arc>` entries. The `<mse_arc>` establishes a collection of HSE stream segments as a single entity,

and defines the nodes which connect to this arc. The allowed environments in a <mse_arc> are <hse_arcs>, <node_source>, <node_sink>, with the following meanings:

- <hse_arcs> - a list of valid integer HSE segment ID's
- <node_source> - a single quoted string listing the name of a mse_node at the head of this mse_arc
- <node_sink> - a single quoted string listing the name of a mse_node at the tail of this mse_arc

The XML entries for the <mse_arc> attributes and environments are shown in Table 14.2.1.

environment	attribute	meaning
<mse_arc>	 name capacity	 required arc name optional flow capacity
<hse_arcs>		list of HSE segments ID's
<node_source>		source node name
<node_sink>		sink node name

Table 14.2.1. <mse_arc> XML

14.3 Dummy Arc XML

A dummy <mse_arc> should not have any internal XML environments, it should only define <mse_arc> name and capacity attributes as shown in the example below.

```
<mse_arc name="R1_endflow" capacity="1400"> </mse_arc>
```

14.3.1 MSE Node

The <mse_nodes> environment can contain the following three entries:

- <mse_node> - endpoints of all <mse_arc>
- <flow_source> - string of <mse_node> name which is the flow source of the entire network
- <flow_sink> - string of <mse_node> name which is the flow sink of the entire network

There can only be one <flow_source> and one <flow_sink> entry. There must be multiple (at least 2) <mse_node>.

A mse_node must exist at the junction and endpoint of every <mse_arc>. Each mse_node entry must correspond exactly in name="" attribute to either a <node_source> or <node_sink> defined in the <mse_arcs>. The mse_node attributes are assigned in order to provide operational parameters and specifications relevant to flow control structures. The mse_node may contain the following environments:

- <open> - operational stage for gate full open ‡
- <close> - operational stage for gate full closed ‡
- <supply> - node supply (excess flow) ‡
- <demand> - node demand (deficit flow) ‡

The XML entries for the <mse_node> environment are shown in Table 14.3.1.

environment	attribute	meaning
<mse_node>		
	name	required node name
	priority	reserved for future use
	purpose	"ws", "fc", "wsfc", "none" †
	label	any string
	designCap	design flow capacity
	structure	"yes" or "no"
	managed	"yes" or "no"
	watermover	"none" or watermover label
<open>		gate open stage ‡
	name	user defined label
<close>		gate close stage ‡
	name	user defined label
<supply>		node supply ‡
	name	user defined label
<demand>		node demand ‡
	name	user defined label

Table 14.3.1. <mse_node> XML

†Attributes for purpose may also be expressed as: "watersupply", "floodcontrol", "watersupplyfloodcontrol".

‡The <open>, <close>, <supply>, and <demand> environments can hold one of <dss>, <const>, or <rc> environments to specify a timeseries file, constant value, or rulecurve timeseries respectively.

14.4 Dummy Node XML

A dummy <mse_node> should not have any internal XML environments, it should only define <mse_node> name attribute as shown in the example below.

```
<mse_node name="S11"></mse_node>
```

14.4.1 MSE Unit

The mse_unit aggregates one or more mse_arc into WCU's. The allowed environments in a <mse_unit> are as follows:

- <unit_arcs> - the collection of <mse_arc> that constitute a water control unit (WCU)
- <inlet> - a <mse_node> that flows into the WCU, there may be multiple <inlet>, but only one <mse_node> per <inlet>
- <outlet> - a <mse_node> that WCU discharges from the WCU, there may be multiple <outlet>, but only one <mse_node> per <outlet>
- <maintLevel> - operational maintenance level of the WCU
- <fcLevel> - operational flood control level of the WCU
- <rcptLevel> - operational rule curve level of the WCU
- <localLevel> - operational local level of the WCU

The XML entries for the <mse_unit> environment are shown in Table 14.4.1.

environment	attribute	meaning
<mse_unit>	 name purpose	 required unit name "ws", "fc", "wsfc", "none" †
<unit_arcs>		list of <mse_arc>
<maintLevel>	 name	regional maintenance stage user defined label
<localLevel>	 name	local maintenance stage user defined label
<fcLevel>	 name	flood control stage user defined label
<rcptLevel>	 name	rule curve point stage user defined label
<inlet>	 name	one <mse_node> name user defined label
<outlet>	 name	one <mse_node> name user defined label

Table 14.4.1. <mse_unit> XML

†Attributes for purpose may also be expressed as: "watersupply", "floodcontrol", "watersupplyfloodcontrol".

‡The <maintLevel>and <localLevel> environments can hold one of <dss>, <const>, or <rc> environments to specify a timeseries file, constant value, or rulecurve timeseries respectively.

If the purpose attribute is explicitly specified in the mse_unit, then the value specified will be the value of the WCU purpose. This will override any other value of purpose that may be set in an outlet node of the WCU. If a value is not specified in the <mse_unit> purpose attribute, then a logical OR operation is carried on all of the <outlet> <mse_node> purpose values. The numeric values of the purpose attributes are shown in Table 14.4.1.

purpose	value
none	0x0000
ws, watersupply	0x0001
fc, floodcontrol	0x0010
wsfc, watersupplyfloodcontrol	0x0011

Table 14.4.1. Value of `mse_unit` and `mse_node` `purpose` attribute.

For example, if an `<mse_unit>` does not specify the `purpose` attribute, and there are three outlet `<mse_node>` which have been assigned a `purpose` of `floodcontrol`, `floodcontrol` and `watersupply` respectively, then the `<mse_unit>` `purpose` will have the value `watersupplyfloodcontrol` (0x0011).

14.5 MSE Network Flat File

In addition to the XML specification of the MSE Network, it is also possible to define the MSE Network representation in a flat-file format. The file format consists of three sections, of which only the first is required. The first section defines a list of arcs (2 nodes) with capacity, and an optional arc label on each line. The second section assigns supply or demand constraints to the individual nodes. The third section defines network-wide properties such as the source node, sink node, and graph flow algorithm push scale. An example of this format is shown below, and resides in the benchmarks BM48 and BM49.

```
// This graph corresponds to the hypothetical flow network from
// the WMM primer for wsneeds, mse_network.xml in BM48 BM63.
//
// Define the network arcs as pairs of nodes with capacity:
//   node1 node2 capacity [label]
S11    S11_A   1400  Reach_1
S11_A  S7      1000  Reach_1S
S11_A  S9       500  Reach_1E
S7     S7_A    1000  Reach_2
// Define the node supply & demand, the sum has to equal zero
//   node +supply/-demand
S11    100
S7    -100
// source, sink and push scale factor
source S11
sink   S7
M      100
```

15 Controller Examples

15.1 Example Geometry

To provide illustrative examples of the controller applications, a sample geometry is defined using a simple canal network. A canal consisting of 4 segments is defined. Each segment is 100m wide, 3535m long, has a bottom elevation of 492m, and a lip height of 0.2m. The first and fourth segments are isolated from the network by junctionblocks. The first segment has a constant segment source, which adds a constant flow to the first segment. The fourth segment has a constant segment sink, which removes a constant flow from the fourth segment. Watermover 1 is attached to segment 1, and is controlled by the first controller. Watermover 2 is attached to segment 4, and is controlled by the second controller. The initial heads in the first and fourth segments are 505m, and 495m respectively. A schematic depiction of the network is shown in Figure 15.

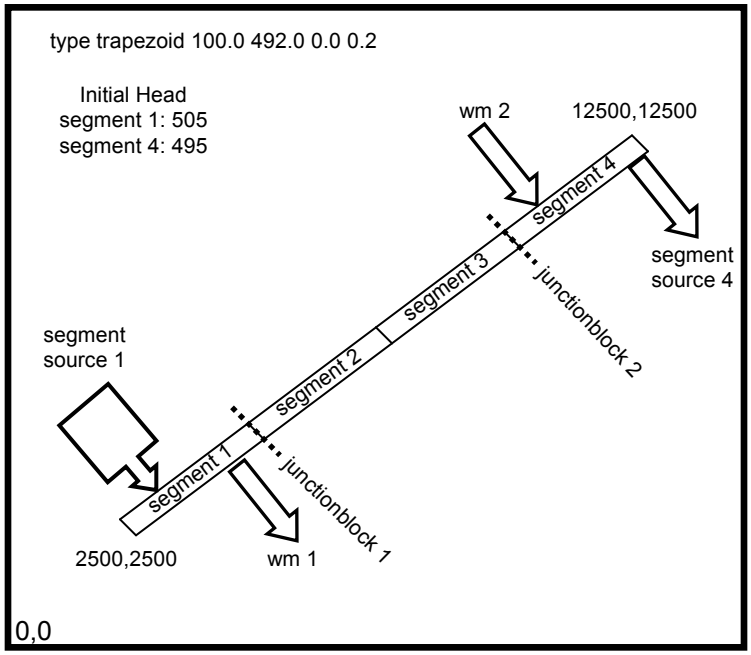

Figure 15: Test Canal Network

15.2 No Control

As a basis for comparison to controlled watermovers, this example shows the 'default' situation with no control applied to the watermovers. The relevant sections of the input XML file are reproduced below:

```
<network>
  <!-- 5 nodes w/ 4 segments -->
  <geometry file="canal3x3.map"> </geometry>
  <initial file="canal3x3.init"> </initial>
  <network_bc>
    <!-- Block canal segments 1&2 and 3&4, 2->3 not blocked -->
    <junctionblock id1="1" id2="2"> </junctionblock>
    <junctionblock id1="3" id2="4"> </junctionblock>
    <!-- Source into segment 1 -->
    <segmentsource id="1"> <const value="1000.0"> </const>
    </segmentsource>
    <!-- Sink from segment 4 -->
    <segmentsource id="4"> <const value="-1000.0"> </const>
    </segmentsource>
  </network_bc>
</network>
<watermovers>
  <!-- discharge from canal segment 1 -->
  <hq_relation wmID="1" id="1" label="">
     <hq>
      0.0    0.0
      495.0  0.0
      499.0 -50.0
      500.0 -150.0
      501.0 -300.0
      510.0 -1000.0
     </hq>
  </hq_relation>
  <!-- inflow into canal segment 4 -->
  <hq_relation wmID="2" id="4" label="">
     <hq>
      0.0    0.0
      490.0  1000.0
      495.0  500.0
      499.0  250.0
      500.0  150.0
      501.0  50.0
      510.0  0.0
     </hq>
  </hq_relation>
</watermovers>
```

This XML file defines two HQ relation watermovers, which in themselves can implement a control function. The first column defines a head elevation, the second column the respective flow value that is executed at the head value. The HQ relation watermover performs linear interpolation for head values between those listed in the first column.

Figure 16 shows the segment 1 and segment 4 response to the watermovers, sources and sinks, without any control exerted on the watermovers. The segment 1 head falls to a value where equilibrium is reached between the segment source and the watermover outflow at a head near 496m, likewise, the segment 4 head rises to an equilibrium near 508m. In order to try and control the canal head one could form the watermover in such a way that the target value is bracketed on either side by balanced positive and negative flow values, in an attempt to establish a self-correcting watermover that would seek a desired equilibrium value.

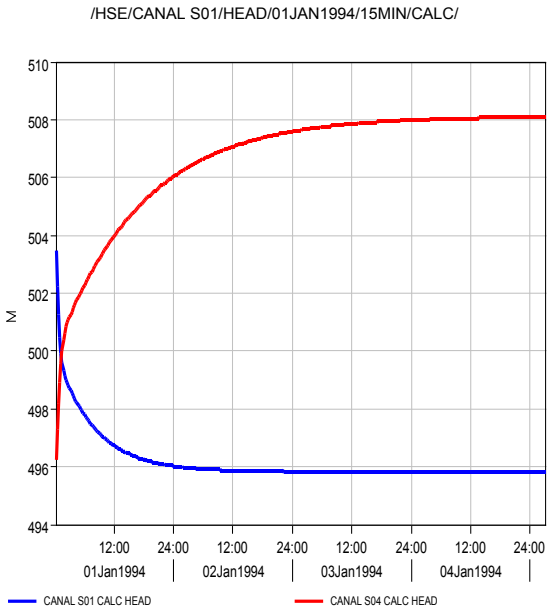

Figure 16: No Control: Head

15.3 PID Controller

Consider now the addition of a PID controller to both watermovers. The PID controller is described in section 4. The controller section of the XML input file is shown below:

```
<controller id="1">
  <!-- Controller for discharge from segment 1 -->
  <pidctrl cid="101" label="PIDCtrl 1: " wmID="1" type="positive"
   Gi="0.01" Gd="0.0" Gp="7.0" ctrlMin="0.0" ctrlMax="1.0" >
    <target label="const_target"><const value="500.0"></const></target>
    <segmentmonitor id="1" attr="head"></segmentmonitor>
  </pidctrl>
  <!-- Controller for pumping into segment 4 -->
  <pidctrl cid="102" label="PIDCtrl 2: " wmID="2" type="negative"
   Gi="0.01" Gd="0.0" Gp="10.0" ctrlMin="0.0" ctrlMax="1.0" >
    <target label="const_target"><const value="500.0"></const></target>
    <segmentmonitor id="4" attr="head"></segmentmonitor>
  </pidctrl>
</controller>
```

Both controllers are set to achieve a target head values of 500m in canal segments 1 and 4. The segmentmonitor identifies the segment from which the state variable (head) is obtained for controller input. The ctrlMin and ctrlMax values set limits on the output of the controller. The controller will compute values of flow regulation which will minimize the error between the state variable and the target value. Figure 17 shows the segment heads resulting from a run of the PID controller. The watermovers have converged to maintenance of the target head values after approximately two days of simulation time.

15.3.1 No Pre-Simulation Iteration

Figure 18 plots the control outputs of the two controllers.

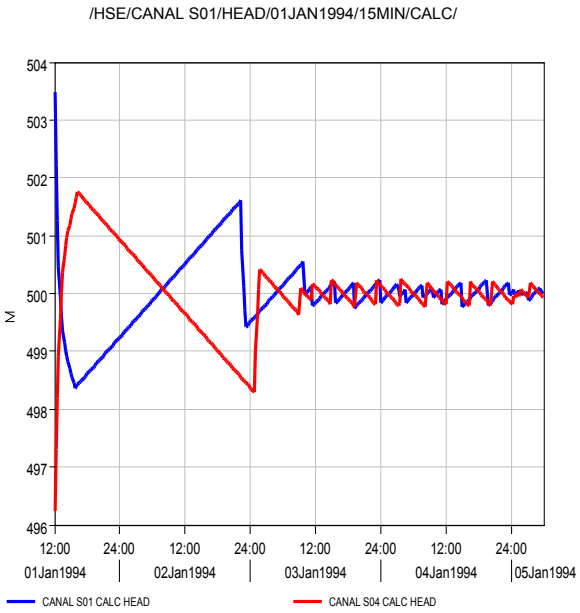

Figure 17: PID No Pre-run Integration: Head

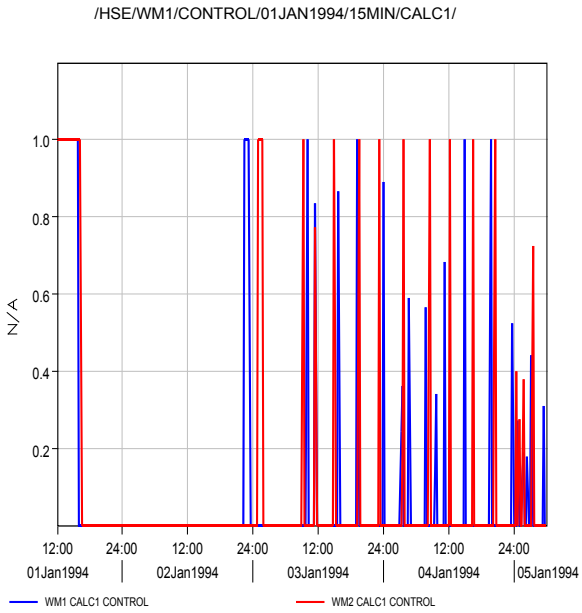

Figure 18: PID No Pre-run Integration: Control

15.3.2 Pre-Simulation Iteration

To allow controllers with integrators (PID, Sigmoid, etc.) to initialize values of error integrals and derivatives, RSM provides the preRunIterations program control variable (Section 10). Figure 19 plots the canal heads under the same conditions as the previous example, but the the controller preRunIterations set to 110. The corresponding control outputs are shown in Figure 20. Both controllers have achieved the steady state target values, and have done so in about half the time required without integration initialization. It is anticipated that careful selection of the controller gains could further improve the response.

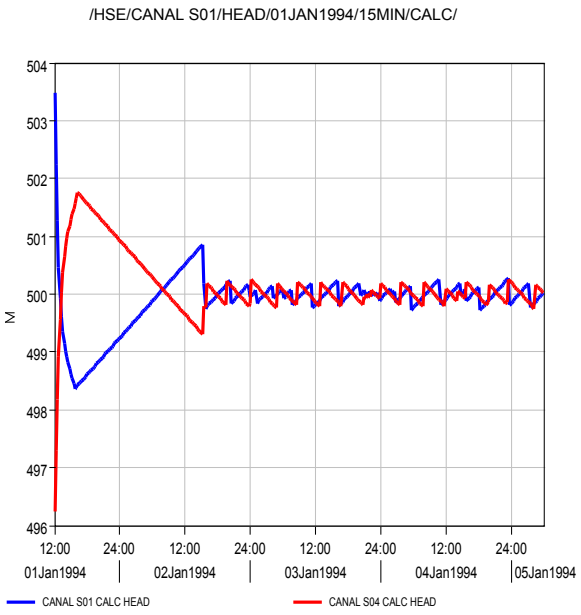

Figure 19: PID With Pre-run Integration: Head

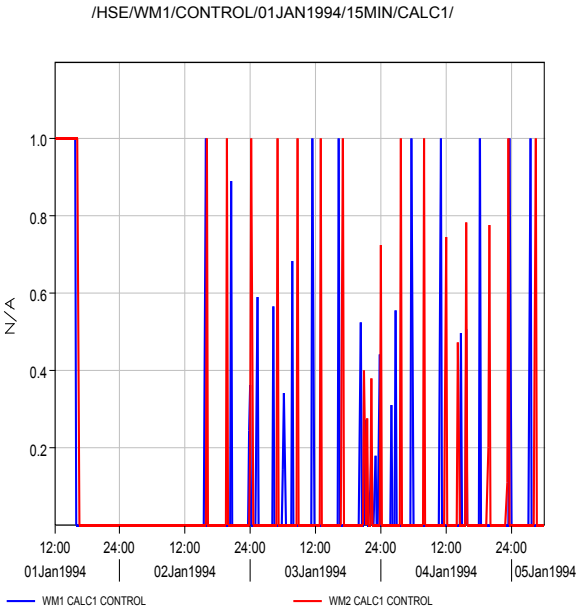

Figure 20: PID With Pre-run Integration: Control

15.4 Sigmoid Controller

The sigmoid controller was also applied to the same canal network and watermovers as the PID controller. The sigmoid controller is detailed in section 5. Following is the controller section from the input XML file:

```
<controller id="1">
  <!-- Controller for discharge from segment 1 -->
  <sigmoidctrl cid="101" label="SigmoidCtrl 1: " wmID="1"
   type="positive" control="on" c="0.1" Gp="10.0" Gi="0.4" >
    <target label="const_target"><const value="500.0"></const></target>
    <segmentmonitor id="1" attr="head"></segmentmonitor>
  </sigmoidctrl>
  <!-- Controller for pumping into segment 4 -->
  <sigmoidctrl cid="102" label="SigmoidCtrl 2: " wmID="2"
   type="negative" control="on" c="0.1" Gp="10.0" Gi="0.4">
    <target label="const_target"><const value="500.0"></const></target>
    <segmentmonitor id="4" attr="head"></segmentmonitor>
  </sigmoidctrl>
</controller>
```

Figure 21 plots the canal segment heads in response to the sigmoid controlled watermovers. The response is seen to be smooth and quickly converging to the target values. The respective controller output values are shown in Figure 22.

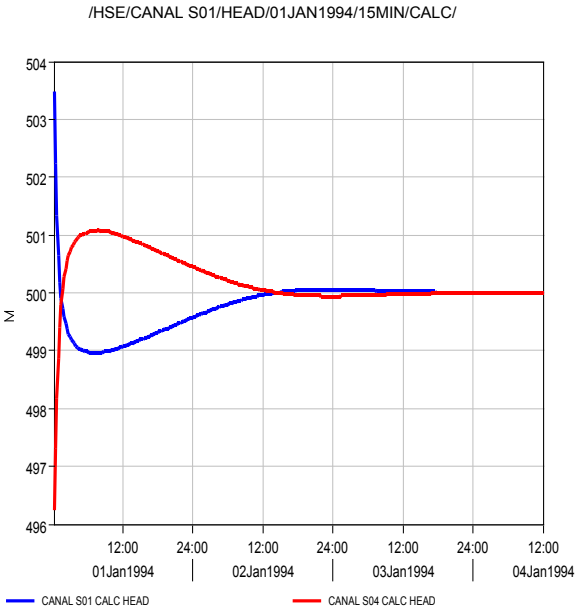

Figure 21: Sigmoid Controller: Head

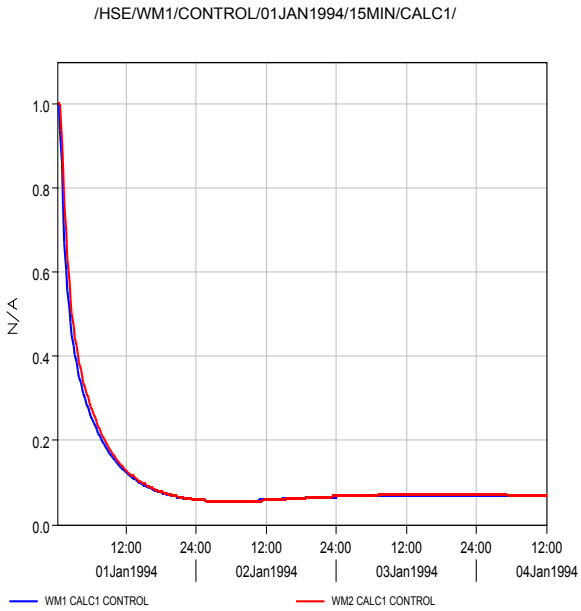

Figure 22: Sigmoid Controller: Control

15.5 SetPoint Controller

The SetPoint controller is detailed in Section 6. Since the setpoint controller does not accept explicit target values, a somewhat different setup in the canal network parameters is used. In particular, the canal segment 1 and canal segment 4 constant segment source/sink are changed from ±1000 to ±500. The HQ relation watermover tables are also different. The corresponding sections for the watermovers and controllers are shown below:

```
<watermovers>
  <!-- discharge from canal segment 1 -->
  <hq_relation wmID="1" id="1" label="">
     <hq>
      0.0    0.0
      490.0 -100.0
      495.0 -500.0
      500.0 -1000.0
      510.0 -1000.0
     </hq>
  </hq_relation>
  <!-- inflow into canal segment 4 -->
  <hq_relation wmID="2" id="4" label="">
     <hq>
      0.0    1000.0
      495.0  1000.0
      500.0  500.0
      510.0  100.0
      600.0  0.0
     </hq>
  </hq_relation>
</watermovers>

<controller id="1">
  <!-- Controller for discharge from segment 1 -->
  <setpointctrl cid="101" label="SPCtrl 1: " wmID="1"
   window="all" setlow="0.0" sethigh="1.0"
   triglow="500.0" trighigh="505.0" trigger="on">
    <segmentmonitor id="1" attr="head"></segmentmonitor>
  </setpointctrl>
  <!-- Controller for pumping into segment 4 -->
  <setpointctrl cid="102" label="SPCtrl 2: " wmID="2"
     window="outside" setlow="0.0" sethigh="1.0" step="down"
     triglow="500.0" trighigh="501.0" trigger="on" >
    <segmentmonitor id="4" attr="head"></segmentmonitor>
  </setpointctrl>
</controller>
```

The controller for canal segment 1 is set to transition between outputs of 0 and 1 at trigger (state variable monitor) values of 500 and 505m. The window type is "all", which indicates that linear interpolation will be performed between the low and high trigger values. When the controller initiates, the head (state variable) is high (Figure 23), and the setpoint controller will limits to the high setpoint of 1 (Figure 24), which will result in a full flow of -1000 for the segment 1 watermover. This causes the segment 1 head to fall into the range between 500 and 505, the linear interpolation range of the setpoint controller. As the head approaches the value of 500m, the controller computes an output value that approaches zero, thereby reducing flow of the watermover to a small value. The transition is a smooth one owing to the linear interpolation of control output values. The controller for segment 4 is set with window "outside", which enacts a binary switch control that will output one of two values, 0 or 1. As shown in Figure 23, the initial head is below 500m, resulting in a control output value of 1. The output is 1 since the type of control is step="down". If the control was set for step="up", the output at this point would have been zero. This allows the watermover to contribute positive flows into the segment which raise the local head. As the head rises to 500.5m (halfway between the triglow and trighigh threshold values, the controller transitions to the zero binary state, thereby turning off the watermover flow. The canal head is then observed to slowly decline as a result of the negative flow segment boundary condition.

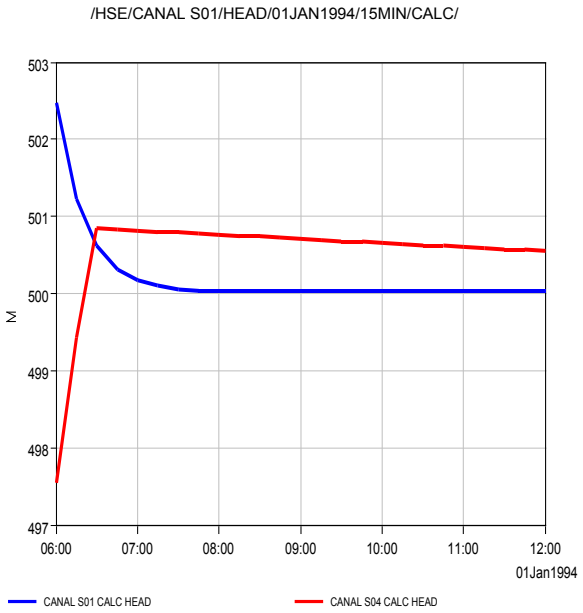

Figure 23: SetPoint Controller: Head

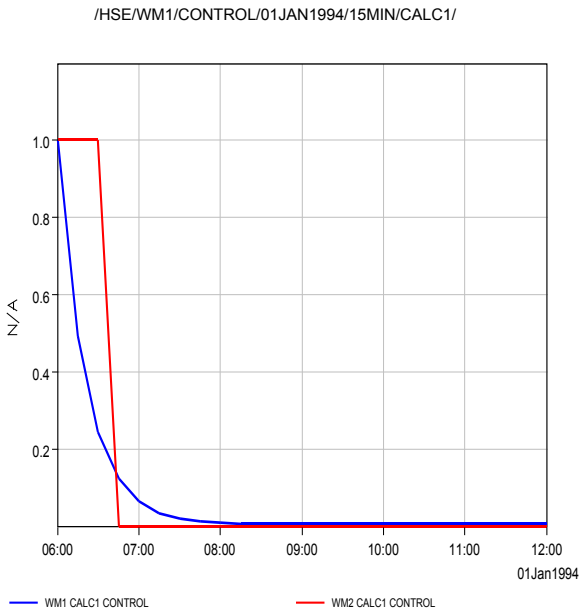

Figure 24: SetPoint Controller: Control

15.6 Sigmoid Gated Weir Controller

This example shows control of a gated weir watermover with a Sigmoid controller. The watermover for canal segment 1 is configured as a gated weir, with a width of 100m, crest elevation at 502.5m, and flow coefficients set to 0.9. Canal segment 1 has a source with an inflow of 1000, canal segment 2 has a segmenthead boundary condition of 500.m applied to maintain the downstream weir head. The XML code for the controller is shown below:

```
<controller id="1">
  <sigmoidctrl cid="101" label="SigmoidCtrl 1: " wmID="1" control="on"
   type="positive" nvals="3" Gi="0.01" Gp="10.0">
    <target label="const_target"><const value="502"></const></target>
    <segmentmonitor id="1" attr="head"></segmentmonitor>
  </pidctrl>
</controller>
```

The canal head elevations and gate control outputs are shown in Figures 25 and 26. The initial head is above the target, so the initial control command is to fully open the gate (control = 1.) so that the canal head falls to a value near the target. The head cannot reach the target as a result of the self- limiting flow imposed by the elevation of the weir crest. Once the head has initially fallen, the controller attempts to partially close the gate in anticipation of not overshooting the target. As the target value is slowly approached, the controller commands the gate to open further to reach the target value.

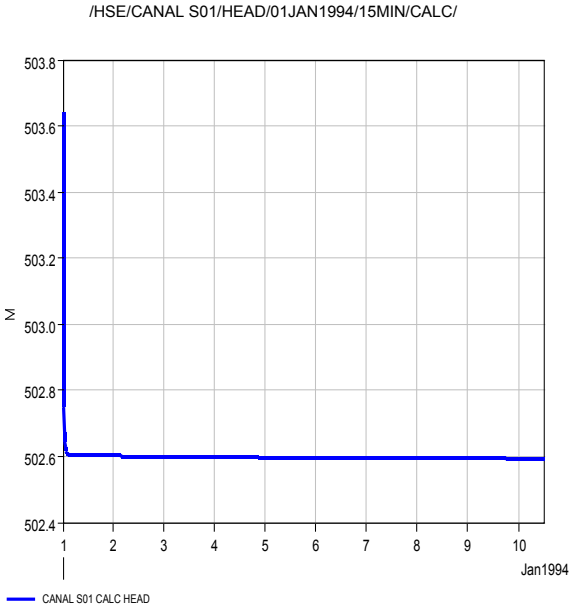

Figure 25: Sigmoid Gated Weir: Head

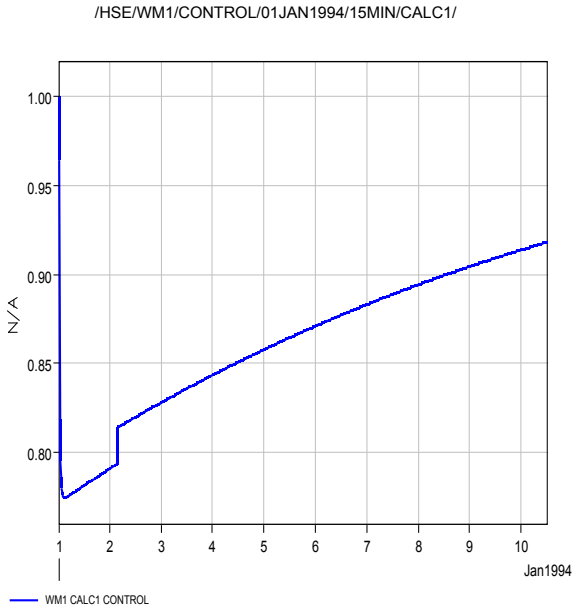

Figure 26: Sigmoid Gated Weir: Control

15.7 Fuzzy Controller

This section provides an example application of the fuzzy controller. The fuzzy controller is described in section 7. The control is implemented on two sets of `hq_relation` watermovers, one set for inflow/discharge from segment one, the other set for inflow/ discharge for segment four.

```
<watermovers>
  <!-- discharge from canal segment 1 -->
  <hq_relation wmID="1" id="1" label="">
     <hq>
       480.0   0.0
       495.0  -100.0
       498.0  -200.0
       499.0  -500.0
       500.0  -1000.0
       501.0  -1000.0
       510.0  -1000.0
     </hq>
  </hq_relation>
  <!-- inflow into canal segment 1 -->
  <hq_relation wmID="2" id="1" label="">
     <hq>
       480.0  1000.0
       495.0  1000.0
       498.0  1000.0
       500.0  1000.0
       505.0  500.0
       510.0  0.0
     </hq>
  </hq_relation>
  <!-- discharge from canal segment 4 -->
  <hq_relation wmID="3" id="4" label="">
     <hq>
       480.0   0.0
       495.0  -100.0
       498.0  -200.0
       499.0  -500.0
       500.0  -1000.0
       501.0  -1000.0
       510.0  -1000.0
     </hq>
  </hq_relation>
  <!-- inflow into canal segment 4 -->
  <hq_relation wmID="4" id="4" label="">
     <hq>
       480.0  1000.0
       495.0  1000.0
```

```
        498.0  1000.0
        500.0  1000.0
        505.0  500.0
        510.0  0.0
      </hq>
    </hq_relation>
  </watermovers>
```

The function of the fuzzy controller is to output control values which enact watermover flows to achieve target head values of 500m in the canal segments 1 and 4. The fuzzy controller is implemented as a dual-input, single-output controller for each watermover.

The XML controller section is reproduced below.

```
<controller id="1">
  <!-- Controller for discharge from segment 1 -->
  <fuzctrl cid="101" wmID="1" fcl="hq2.fcl" label="fcl1"
    <varIn name="segment1Head" monitor="segmentmonitor"
     monID="1" monType="head"> </varIn>
    <varIn name="segment2Head" monitor="segmentmonitor"
     monID="2" monType="head"> </varIn>
    <varOut name="control1Out"> </varOut>
    <segmentmonitor id="1" attr="head"></segmentmonitor>
    <segmentmonitor id="2" attr="head"></segmentmonitor>
  </fuzctrl>
  <!-- Controller for pumping into segment 1 -->
  <fuzctrl cid="102" wmID="2" fcl="hq2.fcl" label="fcl2"
    <varIn name="segment1Head" monitor="segmentmonitor"
     monID="1" monType="head"> </varIn>
    <varIn name="segment2Head" monitor="segmentmonitor"
     monID="2" monType="head"> </varIn>
    <varOut name="control1Out"> </varOut>
    <segmentmonitor id="1" attr="head"></segmentmonitor>
    <segmentmonitor id="2" attr="head"></segmentmonitor>
  </fuzctrl>
  <!-- Controller for discharge from segment 4 -->
  <fuzctrl cid="103" wmID="3" fcl="hq2.fcl" label="fcl3"
    <varIn name="segment3Head" monitor="segmentmonitor"
     monID="3" monType="head"> </varIn>
    <varIn name="segment4Head" monitor="segmentmonitor"
     monID="4" monType="head"> </varIn>
    <varOut name="control1Out"> </varOut>
    <segmentmonitor id="3" attr="head"></segmentmonitor>
    <segmentmonitor id="4" attr="head"></segmentmonitor>
  </fuzctrl>
  <!-- Controller for pumping into segment 4 -->
  <fuzctrl cid="104" wmID="4" fcl="hq2.fcl" label="fcl4"
    <varIn name="segment3Head" monitor="segmentmonitor"
     monID="3" monType="head"> </varIn>
    <varIn name="segment4Head" monitor="segmentmonitor"
     monID="4" monType="head"> </varIn>
    <varOut name="control1Out"> </varOut>
    <segmentmonitor id="3" attr="head"></segmentmonitor>
    <segmentmonitor id="4" attr="head"></segmentmonitor>
  </fuzctrl>
</controller>
```

The `<varIn>` specification specifies the two input variables, as well as the corresponding state variable monitor inputs. Each watermover controller (`cid="101" wmID="1"`) input variable definition specifies the two input variables (e.g. `segment1Head` and `segment2Head`) which derive their respective state information from the head of the corresponding segment monitors.

The specification of the `<varIn>` establishes the information required to link the input state monitor with the input variable in the FCL definition file. Both the `<varIn>` and `<varOut>` variables must have a corresponding entry (`VAR_INPUT`, `VAR_OUTPUT`) in the FCL definition file.

The FCL definition file forms the core of the fuzzy control specification for a fuzzy controller. Refer to the FCL standard [4] for nomenclature and usage. In this example, the FCL file (`fcl="hq2.fcl"`) defines the dual-input, single-output fuzzy controller applied to each controlled watermover, and is reproduced below.

The `VAR_INPUT` and `VAR_OUTPUT` definitions correspond to the `<varIn>` and `<varOut>` definitions in the XML file. The `FUZZIFY` sections delineate the input membership functions which fuzzify the input state variable information. The `DEFUZZIFY` block specifies the output membership function, which in the example is a set of singleton values. The `RULEBLOCK` section defines the rule-base that is applied via the fuzzy logic inferencing method to arrive at a control output value. Each section is described below.

```
FUNCTION_BLOCK Fuzzy_FB
VAR_INPUT
        // input variables
        segment1Head : REAL;
        segment2Head : REAL;
        segment3Head : REAL;
        segment4Head : REAL;
END_VAR
VAR_OUTPUT
        // output variable
        control1Out : REAL;
END_VAR
FUZZIFY segment1Head
        // low ramp, med trapezoid, high ramp
        TERM low  := (499, 1) (500, 0);
        TERM med  := (498, 0) (499, 1) (501, 1) (502, 0);
        TERM high := (500, 0) (501, 1);
END_FUZZIFY
FUZZIFY segment2Head
        // low ramp, med trapezoid, high ramp
        TERM low  := (499, 1) (500, 0);
        TERM med  := (498, 0) (499, 1) (501, 1) (502, 0);
        TERM high := (500, 0) (501, 1);
END_FUZZIFY
FUZZIFY segment3Head
        // low ramp, med trapezoid, high ramp
        TERM low  := (499, 1) (500, 0);
        TERM med  := (498, 0) (499, 1) (501, 1) (502, 0);
        TERM high := (500, 0) (501, 1);
END_FUZZIFY
FUZZIFY segment4Head
        // low ramp, med trapezoid, high ramp
        TERM low  := (499, 1) (500, 0);
        TERM med  := (498, 0) (499, 1) (501, 1) (502, 0);
        TERM high := (500, 0) (501, 1);
END_FUZZIFY
```

```
DEFUZZIFY control1Out
          // All outputs are singletons
          TERM zero         := 0.;
          TERM quarter      := 0.25;
          TERM half         := 0.5;
          TERM threeQuarter := 0.75;
          TERM one          := 1.;
          ACCU: MAX;
          METHOD: COG;
          DEFAULT:= 0;
          RANGE:= (0, 1);
END_DEFUZZIFY
RULEBLOCK No1
          AND : MIN;
          OR  : MAX;
          ACT : MIN;
          RULE 1: IF segment1Head IS low THEN control1Out IS zero;
          RULE 2: IF segment1Head IS med THEN control1Out IS half;
          RULE 3: IF segment1Head IS high THEN control1Out IS one;
          RULE 4: IF segment1Head IS low AND segment1Head IS med
                     THEN control1Out IS quarter;
          RULE 5: IF segment1Head IS high AND segment1Head IS med
                     THEN control1Out IS threeQuarter;
          RULE 6: IF segment2Head IS low
                     THEN control1Out IS threeQuarter;
          RULE 7: IF segment2Head IS high THEN control1Out IS quarter;
          RULE 8: IF segment4Head IS low THEN control1Out IS one;
          RULE 9: IF segment4Head IS med THEN control1Out IS half;
          RULE 10: IF segment4Head IS high THEN control1Out IS zero;
          RULE 11: IF segment4Head IS low AND segment4Head IS med
                      THEN control1Out IS threeQuarter;
          RULE 12: IF segment4Head IS high AND segment4Head IS med
                     THEN control1Out IS quarter;
          RULE 13: IF segment3Head IS low THEN control1Out IS threeQuarter;
          RULE 14: IF segment3Head IS high THEN control1Out IS quarter;
END_RULEBLOCK
END_FUNCTION_BLOCK
```

15.7.1 Fuzzification

The `FUZZIFY` keyword defines an FCL section which specifies the input variable membership function and terms. In this example, the membership functions contain three terms each, `low`, `med` and `high`, where the `low`and `high` terms are 'ramps', and the `med` term is a trapezoid. The input state monitor values are applied to the ordinate of this membership function, generating a membership value μ for each term of the membership function. Figure 27 plots the input membership function for this example.

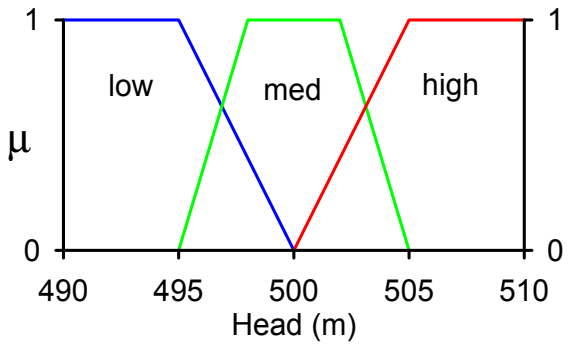

Figure 27: Segment*Head input fuzzification function

15.7.2 Defuzzification

The `DEFUZZIFY` keyword defines an FCL section that specifies the output variable defuzzification membership function and terms. In this example, the membership function has five terms `zero`, `quarter`, `half`, `threeQuarter` and `one`. Each term is a singleton rather than a typical fuzzy membership term. The use of a singleton does not impact the fuzzy inferencing algorithm, or prevent output variable interpolation between output term values. A graph of the output membership function is shown in Figure 28.

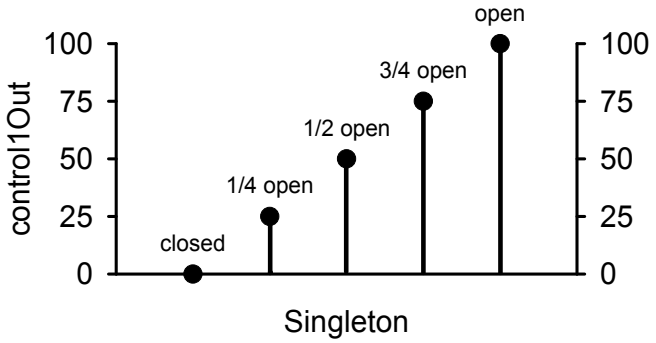

Figure 28: Control1Out output function

15.7.3 Rules

The `RULEBLOCK` section defines the expert rule-base which governs the interaction and response of the controller to the various input states. In this example, a single fuzzy controller rule-base definition is used for both watermover controllers. The rule-base incorporates 10 rules, 5 of which are enacted for each watermover. It would have been equally valid to create two separate FCL files, with the 5 rules for each controller in each FCL file.

The canal head elevations and fuzzy controller control outputs are shown in Figures 29 and 30.

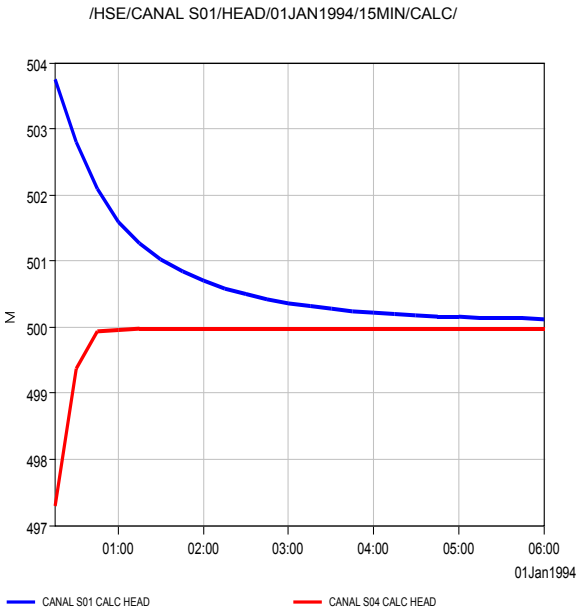

Figure 29: Fuzzy Control: Head

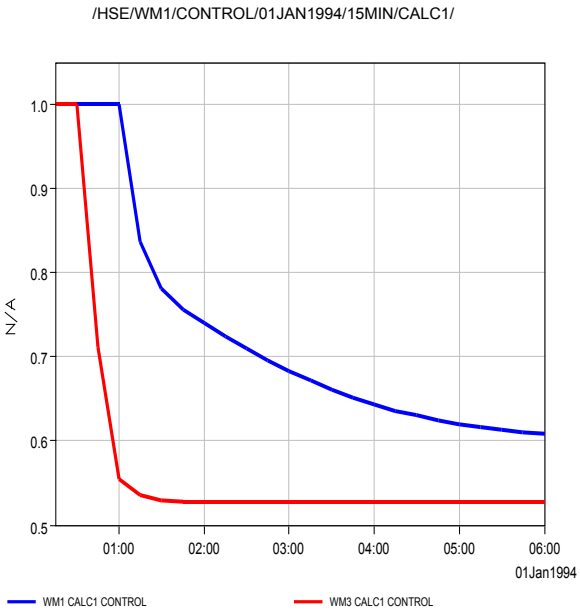

Figure 30: Fuzzy Control: Control

15.8 User Controller

This section provides an example application of the user defined controller. The user controller is described in section 8. The controlled watermovers for segment 1 and segment 4 are shown below.

```
<watermovers>
  <!-- discharge from canal segment 1 -->
  <hq_relation wmID="1" id="1" label="">
     <hq>
      0.0    0.0
      495.0  0.0
      499.0 -50.0
      500.0 -150.0
      501.0 -300.0
      510.0 -1000.0
     </hq>
  </hq_relation>
  <!-- inflow into canal segment 4 -->
  <hq_relation wmID="2" id="4" label="">
     <hq>
      0.0    0.0
      490.0  1000.0
      495.0  500.0
      499.0  250.0
      500.0  150.0
      501.0  50.0
      510.0  0.0
     </hq>
  </hq_relation>
</watermovers>
```

To control the watermovers to achieve target canal levels of 500m, two controllers are defined as shown in the XML controller section reproduced below. Each controller is a user defined function, both functions are contained in the shared object `UserCtrl.so`. Each function will receive an `InputStateMap` which contains two `InputState` class objects, one for the `varIn` `Segment1` and one for `Segment4`. The values of the `segmentmonitor` for each `varIn` will be placed in the `InputState` class object by RSM. The return value of the function will be applied as an amplitude multiplier to the watermover flow value.

```
<controller id="1">
  <!-- Controller for discharge from segment 1 -->
  <userctrl cid="101" label="Segment 1 Ctrl " wmID="1"
   libType="C++" module="./UserCtrl.so" func="Segment1_Control">
     <varIn name="Segment1"  monitor="segmentmonitor"
      monID="1" monType="head"></varIn>
     <varIn name="Segment4"  monitor="segmentmonitor"
      monID="4" monType="head"></varIn>
     <segmentmonitor id="1" attr="head"></segmentmonitor>
     <segmentmonitor id="4" attr="head"></segmentmonitor>
  </userctrl>
</controller>
<controller id="2">
  <!-- Controller for pumping into segment 4 -->
  <userctrl cid="102" label="Segment 4 Ctrl" wmID="2"
   libType="C++" module="./UserCtrl.so" func="Segment4_Control">
     <varIn name="Segment1"  monitor="segmentmonitor"
      monID="1" monType="head"></varIn>
     <varIn name="Segment4"  monitor="segmentmonitor" monID="4"
      monType="head"></varIn>
     <segmentmonitor id="1" attr="head"></segmentmonitor>
     <segmentmonitor id="4" attr="head"></segmentmonitor>
  </userctrl>
</controller>
```

The C++ source code that defines the two controllers is shown below.

```
// userctrl.cc
//
// Example shared object module for User Defined Controller.
// This file is compiled into a shared object module which is loaded
// at runtime by the user specified userctrl section of the hse xml.
//
// To compile this code into an object named UserCtrl.so:
//         gcc userctrl.cc -BSymbolic -shared -o UserCtrl.so
//
// The functions are declared as: extern "C" to prevent name mangling.
// Otherwise, the func= tag in the xml file will specify a name for
// the function that doesn't match that in the shared lib module.
//
using namespace std;

#include <cstdio> // include if you want to use C-style printf
#include <map>
#include "../src/ControllerInput.h" // THIS FILE MUST BE INCLUDED
```

```
//---------------------------------------------------------------------
// Function Segment1_Control for control of watermover into segment 1
//
extern "C" double Segment1_Control(
  map<string, InputState*> *lpInputStateMap ) {

  // define a local map object, assign content from the map pointer
  map<string, InputState*> inputStateMap = *lpInputStateMap;

  double controlOut = 0.;    // output control value
  InputState *lpInputState;  // pointer for access convenience

  // Ensure that we are getting what we expect
  if ( not inputStateMap.size() ) {
    printf("ERROR: Segment1_Control() empty inputStateMap \n");
    return controlOut;
  }
  if ( inputStateMap.find("Segment1") == inputStateMap.end() ) {
    printf("ERROR: Segment1_Control() did not find entry for
            %s in inputStateMap  \n", "Segment1");
    return controlOut;
  }

  // Get pointer to desired inputState struct for this variable
  lpInputState = inputStateMap["Segment1"];
  // Ensure that the pointer is valid
  if ( not lpInputState ) {
    printf("ERROR: SetWM1Controller() Failed to get inputStateMap
            member %s\n", "Segment1");
    return controlOut;
  }

  // Get the current state value for this variable
  double segment1Head = lpInputState->stateIn;

  // Provide control function based on input state variable
  if ( segment1Head > 502. )      controlOut = 1.;
  else if ( segment1Head > 501. ) controlOut = 0.5;
  else if ( segment1Head > 500. ) controlOut = 0.2;
  else                            controlOut = 0.;

  return controlOut;
}
```

```
//---------------------------------------------------------------------
// Function Segment4_Control for control of watermover out of segment 4
//
extern "C" double Segment4_Control(
  map<string, InputState*> *lpInputStateMap ) {

  // define a local map object, assign content from the map pointer
  map<string, InputState*> inputStateMap = *lpInputStateMap;

  double controlOut = 0.;    // output control value
  InputState *lpInputState;  // pointer for access convenience

  // Ensure that we are getting what we expect
  if ( not inputStateMap.size() ) {
    printf("ERROR: Segment4_Control() empty inputStateMap \n");
    return controlOut;
  }
  if ( inputStateMap.find("Segment4") == inputStateMap.end() ) {
    printf("ERROR: Segment4_Control() did not find entry for
            %s in inputStateMap  \n", "Segment4");
    return controlOut;
  }

  // Get pointer to desired inputState struct for this variable
  lpInputState = inputStateMap["Segment4"];
  // Ensure that the pointer is valid
  if ( not lpInputState ) {
    printf("ERROR: SetWM1Controller() Failed to get inputStateMap
            member %s\n", "Segment4");
    return controlOut;
  }

  // Get the current state value for this variable
  double segment4Head = lpInputState->stateIn;

  // Provide control function based on input state variable
  if ( segment4Head < 498. )      controlOut = 1.;
  else if ( segment4Head < 499. ) controlOut = 0.5;
  else if ( segment4Head < 500. ) controlOut = 0.2;
  else                            controlOut = 0.;

  return controlOut;
}
```

The canal head elevations and user controller control outputs are shown in Figures 31 and 32.

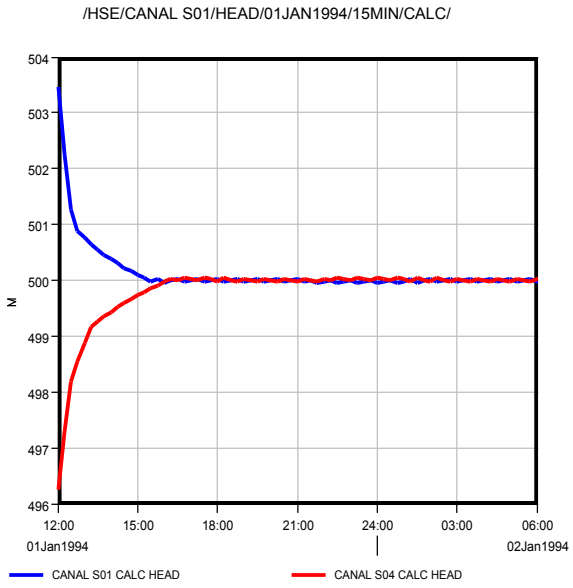

Figure 31: User Control: Head

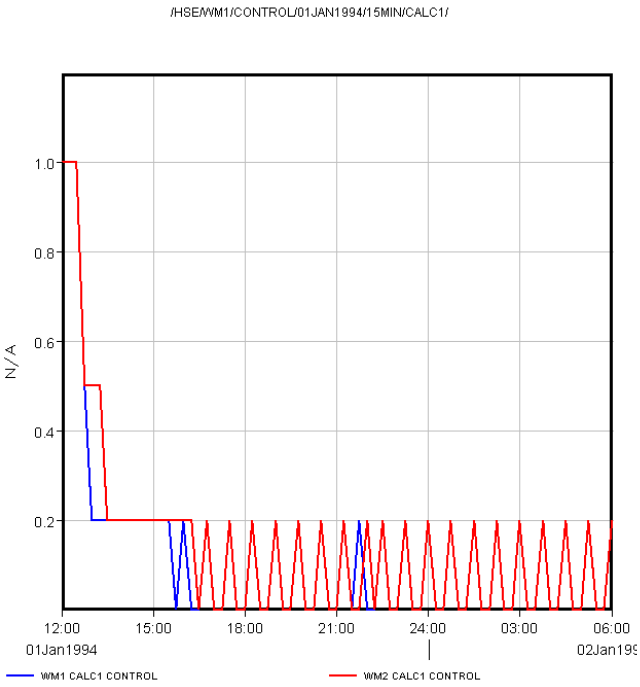

Figure 32: User Control: Control

References

[1] Lal, Wasantha,
Weighted implicit finite-volume model for overland flow,
ASCE *Journal of Hydraulic Eng.*, 124(9), Sep 1998, pp. 941-950

[2] Regional Simulation Model (RSM) User's Manual, Management Simulation Engine (MSE) Supervisors, Office of Modeling
Model Development Division (4540)
South Florida Water Management District
3301 Gun Club Road, West Palm Beach, FL

[3] Fuzzy Control Library
Description and Application Programming Interface
December 10 2003
Office of Modeling
Model Development Division (4540)
South Florida Water Management District
3301 Gun Club Road, West Palm Beach, FL See the PDF file FuzzyControlAPI.pdf from CVS:
`CVSROOT/models/hse/doc/fcll`

[4] International Electrotechnical Commission (IEC)
Technical Committee No. 65
Industrial Process Measurement and Control
Sub-committee 65B: Devices
IEC 1131 - Programmable Controllers
Part 7 - Fuzzy Control Programming
See the PDF file FuzzyControlStandard.pdf from CVS:
`CVSROOT/models/hse/doc/fcll`

[5] Regional Simulation Model (RSM) User's Manual, Hydrologic Simulation Engine (HSE) Components, Office of Modeling
Model Development Division (4540)
South Florida Water Management District
3301 Gun Club Road, West Palm Beach, FL

[6] Ford, L. R., Fulkerson, D. R., Flows in Networks, Princeton University Press, 1962

[7] Ahuja, R. K., Magnanti, T. I., Orlin, J. B., Network Flows: Theory, Algorithms, and Applications, Prentice Hall, 1993