

Technical Documentation Version 8.0

Solution Approaches



Center for Advanced Decision Support for Water and Environmental Systems (CADSWES)

UNIVERSITY OF COLORADO **BOULDER**

These documents are copyrighted by the Regents of the University of Colorado. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means electronic, mechanical, recording or otherwise without the prior written consent of The University of Colorado. All rights are reserved by The University of Colorado.

The University of Colorado makes no warranty of any kind with respect to the completeness or accuracy of this document. The University of Colorado may make improvements and/or changes in the product(s) and/or programs described within this document at any time and without notice.

Contents

1. Simulation 1
- Mathematical Basis for Reservoir Simulation 1
- Dispatch Methods 2
  - User Method-dependent Dispatch Methods 2
- Slot Values 3
  - Set Value 3
  - Object's Response to Setting a Slot Value 4
  - Iteration 4
- Simulation Controller 4
  - Initialization 5
    - First Dispatch Timestep 5
    - Initialization Rules 6
  - Time Horizon Simulation 6
  - Underdetermination 6
- Object Dispatching Patterns 7
  - Two Reservoirs With Tailwater-dependent Energy 7
  - Bidirectional Canal 7
  - Reach and AggDiversionSite Interaction 8

2. Rulebased Simulation 9
- Simulation Review 9
- How Rulebased Simulation Works 10
- Rules and Rulesets 11
- Rule Execution 12
- Rule Dependencies 13
- Rules Agenda 14
- Rulebased Simulation Controller 15
  - Controller Priority 16
  - Simulation/Rules Interaction 16
  - Timestep Dependence of Rules 18
  - Run Cycles 18
- Slot Priorities and Flags 19
- Resetting Slot Values 21
  - Multi Slot Behavior 22
  - Link Behavior 24
- Dispatching 24
  - Governing Slots 24

Determination of Dispatch Method 27

3. Multiple Run Management 31

About Multiple Runs 31

Running a Preconfigured MRM Model 31

Managing MRM Configurations 32

Creating a New Configuration 33
Editing an Existing Configuration 33
Deleting an Existing Configuration 33
Copying an Existing Configuration 33

Setting Up a Multiple Run Configuration 34

Name 35
Description 35
Mode 36
Concurrent Runs 36
Consecutive Runs 37
Iterative Runs 38
Policy 41
Input 42
Initialization DMI 43
Input DMI Runs 43
Index-sequential Runs 44
Combined Runs 47
Ensembles 47
Ensemble Configuration 48
Ensemble Metadata 51
Output 52
DMI 52
RDF 52
CSV Files 54
NetCDF Files 57

Saving and Restoring Initial Model State 59

Distributed Concurrent Runs 60

User Interface Overview 62
MRM Configuration 62
Remote Manager and Status Dialog 63
How to Make a Distributed Run 65
Creating or Changing a Configuration 65
Rerunning a Configuration 66
How It Works 66
RiverWare Remote Manager 67
XML Configurations 67

RDF Viewer 69

Accessing the RDF Viewer 70
How to Use the RDF Viewer 71

4. Accounting 73
5. Optimization 75

Contents

Chapter 1
Simulation

The Simulation Controller manages the execution of a run, including when objects solve and how they solve. As objects solve, the effects of their solutions propagate to other objects. When an object has enough information to solve, we say it is ready to *dispatch*.

Topics

- Mathematical Basis for Reservoir Simulation, page 1
- Dispatch Methods, page 2
- Slot Values, page 3
- Simulation Controller, page 4
- Object Dispatching Patterns, page 7

Mathematical Basis for Reservoir Simulation

Using RiverWare requires knowledge of elementary reservoir modeling mechanics. Reservoir modeling in RiverWare is accomplished using a mass balance approach. The equation for reservoir mass balance is as follows:

$$Storage_t = Storage_{t-1} + \Sigma(Inflows \times \Delta t) - \Sigma(Outflows \times \Delta t) + Gains - Losses$$

For this section, it is important to more fully understand the Outflows in the above equation. Outflow in RiverWare, is expressed as follows:

$$Outflow = Release(s) + Spill(s)$$

Along with the greater detail in the Outflow term of the basic mass balance equation, Total Inflow in RiverWare is expressed as follows:

$$TotalInflow = Inflow + HydrologicInflow$$

Further refinements may also be made to these equations and the Gain and Loss terms. Some of these include Evaporation, Precipitation, Bank Storage, Pumped Storage Flow, Seepage, Diversion, Return Flow, and Canal Flow.

Given any two of Inflow, Outflow or current Storage, the third variable may be solved for, when the previous Storage is known. Storage can also be determined from the Pool Elevation. Further, the Release (or Energy) and Spill could be specified (together) to determine Outflow.

The discretization of the values in these timeseries is important to recognize. The Inflow and Outflow slots contain the values for the average flow during each timestep, while the Storage and Pool Elevation slots contain the values

at the end of each timestep. In other words, the flow values are pulse data, while the elevation and storage values are instantaneous data. Values in RiverWare slots are only as precise as the timestep length used to generate them.

Dispatch Methods

Each object in RiverWare has one or more Dispatch Methods. The Dispatch Methods represent the various combinations of inputs and outputs which are valid states for solving the object's physical process equations. Each Dispatch Method has a list of Dispatch Conditions, a set of slots with required known values and slots with required unknown values. When the required knowns and unknowns are both satisfied, the Dispatch Method can execute. Table 1.1 lists some Dispatch Methods and their conditions for the Level Power Reservoir object.

> **Note:** For all Dispatch Methods to be able to solve, the previous storage must also be known. This is not in the Dispatch Conditions, but is checked when the object solves.

Table 1.1

Method	Knowns	Unknowns
Solve given Inflow, Outflow	Inflow Outflow	Storage Pool Elevation
Solve given Inflow, Storage	Inflow Storage	Pool Elevation Outflow Energy
Solve given Outflow, Storage	Outflow Storage	Pool Elevation Inflow
Solve given Inflow, Pool Elevation	Inflow Pool Elevation	Storage Outflow Energy
Solve given Outflow, Pool Elevation	Outflow Pool Elevation	Storage Inflow
Solve given Outflow, Storage	Outflow Storage	Pool Elevation Inflow
Solve given Energy, Storage	Energy Storage	Pool Elevation Inflow
Solve given Energy, Pool Elevation	Energy Pool Elevation	Storage Inflow
Solve given Energy, Inflow	Energy Inflow	Pool Elevation Storage

User Method-dependent Dispatch Methods

Dispatch Methods are registered with the controller and added to the method table at the beginning of each run. The method table contains a list of all of the potential dispatch methods for an object. Sometimes, the decision as to

which Dispatch Methods to add to the method table depends on the selected User Methods. For example, the Dispatch Method listed in Table 1.2 is only added to the dispatch table when the Solve Hydrologic Inflow user method is selected.

In this case, knowing the Inflow, Outflow and Storage does not make the object overdetermined. As long as the Hydrologic Inflow is unknown, the object can dispatch the method to solve for it.

Table 1.2

Method	Knowns	Unknowns
Solve given Inflow, Outflow, Storage	Inflow Outflow Storage	Pool Elevation Hydrologic Inflow

Slot Values

Slot timesteps and cells which do not have a value appear as “NaN,” or “Not a Number.” When a slot receives a value, it can occur via one of the following mechanisms:

- Direct user input
- A value propagated across a link from another object
- A value set by the object’s user or dispatch methods

Set Value

When one of the three mechanisms attempts to set a value on a slot, the following steps are taken in order:

1. Convergence. If a value already exists in the slot, check for convergence. If the difference between the old value and the new value is within the slot’s convergence criterion, do not set it. See “Configure Slot Dialog” in *User Interface* for details on how to set convergence for a series slot.
2. Max Iterations. If a value already exists in the slot, check max iterations. If the value has already been set as many times as the max iterations criterion, do not set it again. In this case, a warning message is posted to diagnostics. The max iterations setting is available in the Run Control dialog by selecting **View**, then **Simulation Run Parameters** from the workspace; see “Simulation Run Parameters” in *User Interface* for details.
3. Overdetermination. Check for over overdetermination. Overdetermination is detected by examining how the previous slot value was set. If the slot has a previous value, and this previous value was set from a different source, the object is overdetermined. In this case, post an overdetermination error and abort the run.
4. Set Value. If the previous checks succeed, set the (new) value in the slot.
5. Propagation. If the slot is linked, propagate the value across the link(s).

Object's Response to Setting a Slot Value

When a slot is set on an object, the following occurs:

1. Dispatch Slot. Check if the slot which was set is a dispatch slot. Dispatch slots are the subset of SeriesSlots which may be Dispatch Conditions and may be linked to other objects. Only dispatch slots cause objects to redispatch when they are set. If the slot is not a dispatch slot, nothing else is done.
2. Dispatch Conditions. If the object has not yet dispatched this timestep, check the dispatch conditions of each Method on the method table. If a Method's conditions are satisfied, this Method is used for the dispatch. If no Method's conditions are satisfied, the object must wait for more information; nothing else is done. If the object has already dispatched this timestep, redispatch with the same Method.
3. Notify Controller. If a valid Dispatch Method was found in the previous step, notify the controller of the Method to be dispatched. The controller then puts the object on the dispatch queue.

Iteration

After an object has dispatched during a timestep, it will dispatch again if *any* dispatch slot is reset. When a dispatch slot's value changes, there is a high probability that the previous solution for this object at this timestep is now invalid. Since all values are known, the object cannot match any of its Dispatch Methods' Dispatch Conditions (remember that Dispatch Conditions include required unknowns as well as required knowns). The object redispatches using the same Dispatch Method that was invoked previously.

Simulation Controller

The Simulation Controller is responsible for maintaining the order of execution at the highest level of simulation. The steps, in order, are as follows.

1. Initialization for simulation
   a. Clear all output and values from previous runs for all timesteps in all series slots.
   b. Set user inputs.
   c. Propagate user inputs across link.
   d. Determine first dispatch timestep. See "Initialization" on page 5 for details.
2. Execute rules in the Initialization Rules RPL set. See "Initialization Rules" on page 6 for details.
3. Simulation Beginning of Run
   a. Execute Beginning of Run methods for all objects.
   b. Evaluate Beginning of run expression slots for all timesteps.
4. For each timestep:
   a. Set the controller clock to the timestep time.
   b. Execute Beginning of Timestep methods for all objects.

c. Evaluate Beginning of timestep, current timestep only expression slots

d. Dispatch objects until the queue is empty, simulating the effects of the user inputs and default values.

e. Execute End of Timestep methods on all objects.

f. Evaluate end of timestep, current timestep only Expression slots

5. Execute End of Run Simulation methods on all objects.

6. Evaluate End of Run expression slots.

Initialization

During Initialization of the run, the controller first resets all output slots to NaN, registers input values, and propagates values across links. Then, it finds the first dispatch timestep for each object.It then executes Initialization rules.

First Dispatch Timestep

Next, the controller determines the first timestep at which an object is allowed to dispatch. For the majority of objects, this is the start timestep. For Reaches, Control Points, and Confluences objects, it is sometimes necessary to allow dispatching to take place during prerun timesteps. This enables prerun inputs to route downstream thus allowing the system to solve on the start timestep.

Finding Earliest Upstream Inputs

For Reaches, Control Points, Confluences, Inline Power Plants, and Stream Gages, the first dispatch timestep is determined by looking for the earliest Input value on the Inflow slot (Inflow1 or Inflow2 on the Confluence). Then, the method travels upstream to the linked object (or objects in the case of a confluence) and looks for the earliest input value. This search process continues until either the top of the basin is met or a non-Reach, Control Point, Confluence, Inline Power Plant, or Stream Gage object is met.

Note: The search stops if it finds a reach with either the Kinematic, Muskingum Cunge, or MacCormack routing method selected. The first dispatch timestep is the earliest at which the object is able to dispatch; dispatching is still controlled by the object's dispatch conditions.

Note: The search only looks for input values ("I" or "Z") that are known before the start button is pressed. Initialization rules are executed later, so values set by those rules are not recognized as inputs.

Initialize Flow Slots for Routing

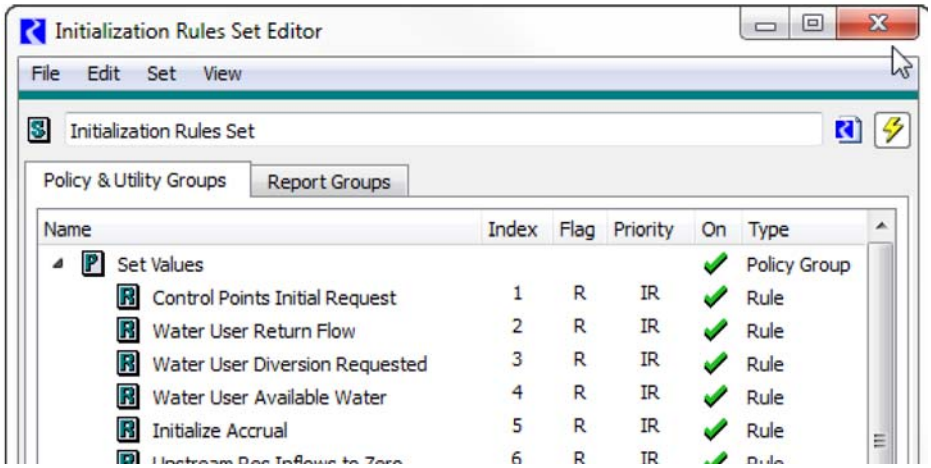
In addition to the search for earliest upstream inputs, methods in the Initialize Flow Slots for Routing category on the Computational Subbasin can be used to determine and set the number of prerun timesteps necessary for downstream lagging. See "Initialize Flow Slots for Routing" in *Objects and Methods* for details.

When these methods are selected, the earliest dispatch timestep is determined by also searching downstream to add up the required number of timesteps based on the selected Reach routing methods. See "Initialize Flow Slots for Routing" in *Objects and Methods* for details on these methods.

Initialization Rules

Initialization Rules are a set of RPL rules associated with the model which can be executed as part of run initialization to set up data for the run. Initialization Rules provide a complement to user inputs and execution of DMIs.

See "Initialization Rules Set" in *RiverWare Policy Language (RPL)* for details on Initialization Rules.



Time Horizon Simulation

Each object has its own clock. The controller clock, which is shown in the Run Status dialog, is not necessarily an indication of the timestep at which the objects are dispatching. Simulation proceeds without regard to the controller clock, except for executing the next Beginning of Timestep when the dispatch queue is empty.

Time horizon dispatching allows the system to solve over several timesteps, depending on which values are known. Due to time lags between reach Inflows and Outflows, objects may not have enough information to dispatch at a given timestep until later in the simulation. Over an entire river basin simulation, this can result in dispatching of linked objects at different timesteps.

When a model's objects have enough information to dispatch later timesteps at the start of the simulation, they will. In such models, most of the dispatching can take place while the controller clock is still on the first timestep. In these cases, the Run Status dialog shows the controller clock on the first simulation timestep for most of the execution time. Since most of the timesteps solve, the simulation proceeds very rapidly through the other timesteps.

> **Note:** The last timestep at which objects dispatch is typically the Finish Timestep but can be specified by the user; see "Simulation Run Parameters" in *User Interface* for details. This allows models that lag or forecast to correctly compute values near the end of the run.

Underdetermination

When objects do not dispatch during the run, no warning or error is generated. Lack of sufficient data to force dispatching is not considered an error. The Dispatch Information dialog should be reviewed after each run to verify that all objects dispatched for all timesteps.

Object Dispatching Patterns

Although a model may solve in many different ways, depending on its inputs and outputs, object types often dispatch in predictable ways. Several *patterns* of dispatching emerge from this modeling approach.

Two Reservoirs With Tailwater-dependent Energy

Two Reservoirs whose states are interdependent will alternately dispatch until they reach a stable solution. The most common example of this inter-reservoir iteration is an upstream reservoir with an Energy request whose tailwater elevation is dependent on a downstream Reservoir. The Turbine Release to meet the Energy request becomes the Inflow to the lower Reservoir. If Outflow is known, the lower Reservoir solves for a new Pool Elevation, which influences the Tailwater Elevation and Operating Head of the upstream reservoir. The iteration proceeds as follows:

1. The upstream Reservoir starts the iteration by dispatching with Solve given Energy, Inflow and calculating an Outflow required to meet the Energy request. This initial value is based on incomplete tailwater data. The value propagates across a link to the downstream Reservoir.
2. The upstream and downstream Reservoirs iterate with the following steps until they converge:
   a. The downstream Reservoir dispatches with Solve given Inflow, Outflow and calculates a new Storage and Pool Elevation based on the Outflow from the upstream Reservoir (Inflow) and its user-input Outflow. This Pool Elevation propagates across a link to the Tailwater Base Value of the upstream Reservoir.
   b. The upstream Reservoir dispatches with Solve given Energy, Inflow and solves for a new Turbine Release to meet the Energy request based on the backwater effects of the downstream Reservoir (Tailwater Base Elevation) and resulting change to its Operating Head. The Outflow value propagates across a link to the downstream Reservoir.

Bidirectional Canal

The Canal object behaves differently from other objects in RiverWare. Due to instability of three object iterations, the group of objects surrounding a canal must be solved simultaneously. The order of execution is as follows.

1. One of the two Reservoirs dispatches with Solve given Inflow, Outflow.
   a. Canal Flow is linked but not known, so the Reservoir sets its current Pool Elevation to the previous timestep's Pool Elevation and exits the dispatch.
2. The other Reservoir dispatches with Solve given Inflow, Outflow.
   a. Canal Flow is linked but not known, so the Reservoir sets its current Pool Elevation to the previous timestep's Pool Elevation and exits the dispatch.
3. The Canal dispatches with Solve given Elevation 1, Elevation 2, which requires the following known values:
   – Elevation 1
   – Elevation 2
   – Previous Elevation 1

   – Previous Elevation 2

4. The Canal then solves a set of up to seven linear equations to determine the flow through the Canal. The flow value, and its inverse, are set on the Flow 1 and Flow 2 slots. These values propagate across links back to the Reservoirs.
5. Both Reservoirs redispatch independently. Since Canal Flow is linked and known, the dispatch can now solve for all of the Reservoir parameters.

See "Solve given Elevation 1, Elevation 2" in *Objects and Methods* for details on the canal dispatch method.

Reach and AggDiversionSite Interaction

Reach objects which are linked to an Aggregate Diversion Site also have a special dispatching order. The amount of flow in the Reach determines the quantity of water available for diversion from the Reach, and the amount actually diverted affects the quantity of Outflow from the Reach. The order of dispatching is as follows.

1. The Reach dispatches with one of two Dispatch Methods because of a known Inflow or Outflow.
   a. If Inflow is known, the Dispatch Method is Solve given Inflow, No Routing. If the Diversion slot on the Reach is linked but not known, the Available for Diversion slot is set equal to the Inflow.
   b. If Outflow is known, the Dispatch Method is Solve given Outflow, No Routing. If the Diversion slot on the Reach is linked but not known, the Available for Diversion slot is set equal to the user-specified minimum Available for Diversion.
2. The Available for Diversion value propagates across the link to the Aggregate Diversion Site.
3. The Aggregate Diversion Site dispatches with Solve Lumped given Total Diversion Request, Solve Sequential given Total Diversion Requested, or simply propagates values across links, depending on the selected Linking Structure. In all cases, a Total Diversion value is determined for the Aggregate Diversion Site. This value propagates across a link to the Diversion slot on the Reach.
4. The Reach can now redispatch and solve for its Inflow or Outflow.

Chapter 2
Rulebased Simulation

Rulebased simulation is an extension of the RiverWare basic Simulation feature, in which some of the data to solve an underdetermined model is provided by a set of user-defined rules. Rule execution alternates with dispatching to simulate the effects of the rules in the model.

Topics

- Simulation Review, page 9
- How Rulebased Simulation Works, page 10
- Rules and Rulesets, page 11
- Rule Execution, page 12
- Rule Dependencies, page 13
- Rules Agenda, page 14
- Rulebased Simulation Controller, page 15
- Slot Priorities and Flags, page 19
- Resetting Slot Values, page 21
- Dispatching, page 24

Simulation Review

Rulebased Simulation can only be understood once the fundamentals of basic Simulation have been mastered. This section assumes that you have a familiarity with simulation; see "Simulation" on page 1 for details. However, we will briefly review some key concepts to ensure that everyone's knowledge is current.

Dispatch Slots

Each object has a set of slots for which new values will cause the object to try to solve. These are the object's dispatch slots. All dispatch slots are time series slots. All slots which can be linked to slots on other objects are dispatch slots, and all slots which are included in the dispatch conditions of any dispatch method are dispatch slots. Whenever a value is set on a dispatch slot, the object on which the slot resides must examine its state to determine if it needs to solve; i.e., execute a dispatch method.

Dispatch Methods

Dispatch methods are the algorithms for solving the physical process equations of RiverWare objects. There is one dispatch method for each combination of inputs and outputs for which a solution is possible.

Dispatching

If an object determines that it can execute a dispatch method, it notifies the controller of the method it will execute. The controller adds the object to a queue of objects waiting to execute their methods. When a dispatch method on an object is executed, we say that the object has dispatched. The objects dispatch in the order they are added to the queue.

Dispatch Conditions

Each dispatch method has a set of dispatch conditions which must be met for that dispatch method to be executed during a timestep. The dispatch conditions specify the input/output data combination required for that method to successfully execute. The dispatch conditions for each method are made up of two slot lists called the *knowns* and the *unknowns*. If all of the required knowns are known and all of the required unknowns are not known, the object may solve using that dispatch method. The dispatch conditions are checked whenever a value is set on a dispatch slot and no dispatch method has yet executed at that timestep.

Redispatching

After an object has dispatched once, if one or more of its dispatch slots gets a new value (for example by link propagation from another object), the object must resolve. In pure simulation, the object simply redispatches with the same dispatch method as the first time it dispatched in the timestep. In other words, an object may solve using only one combination of input/output data on any given timestep.

Over/Underdetermination

Each object has a conflict list which lists a combination of slots which, if all are known, result in too many knowns and not enough unknowns for the object to solve. If this situation is detected during a run, an over-determination error is posted, and the simulation is halted. Also, each slot keeps track of where its values originate: user input, link propagation, or set by the object's methods. At each timestep, each slot can receive information from only one of these three sources. If a slot is set by one source, and subsequently reset by another source, an over-determination error is posted, and the simulation is halted. As a result, values may only propagate across a link in one direction during a single timestep.

How Rulebased Simulation Works

Rulebased Simulation is a variation of RiverWare basic Simulation, in which an underdetermined model is provided additional information from user-specified rules that represent the operating policy of the basin. A Rulebased Simulation model does not contain enough inputs for all objects to solve. Instead, it relies on the rules to provide additional information needed to fully solve the model. Rules provide this information by setting slot values in the model. The information provided by the rules, together with the input data, results in an exactly specified model.

The rules are logical statements formulated by the modeler and written within RiverWare, in a special language which is interpreted at runtime. Thus, the rules are data which can be saved and modified without having to recompile a program.

Rulebased Simulation works by alternating between executing rules and dispatching objects on the queue. The rules set values in slots. As a result of these values, objects may have enough information to dispatch. The ensuing dispatching simulates the effects of the rules in the model.

The Rulebased Simulation controller is available in RiverWare as an alternative controller to the Simulation controller. The Rulebased Simulation controller uses the same physical process algorithms as the Simulation controller, so the same user methods and slots are available regardless of which controller is active. You do not need to modify a model built for basic Simulation to run it under the Rulebased Simulation controller, other than setting slots to "output" which will be set by the rules. Data specific to User Methods do not need to change. This allows the same model to be used for what-if scenario runs using Simulation and for policy-driven model runs with Rulebased Simulation.

Rules and Rulesets

A single rule consists of one or more logical statements which assign values to one or more slots in the model. The rule is written in a special language, the RiverWare Policy Language (RPL). The rule's logic may reference the current state of the model—determined from the values in slots—to decide if, and to what value, slots are assigned.

Rule

A rule is a statement, formulated in the RiverWare Policy Language, which assigns values to one or more series slots in the model based on logic within the statement and possibly based on the values of slots in the model.

> **Note:** Rules can assign values on *series slots only*. If you wish to set values on table slots, use Initialization Rules; see "Initialization Rules Set" in *RiverWare Policy Language (RPL)* for details.

Rules are designed and constructed by modelers to mimic policy and decisions which would normally be implemented in scheduling river and reservoir operations. Ideally, each rule represents a specific operating policy. The set of rules represents the entire operating policy for the basin. In order to resolve conflicting policy in rules, the modeler gives each rule a unique priority relative to the other rules. Higher priority is indicated by a smaller number; i.e., priority #1 is higher than priority #2.

Ruleset

A ruleset is a set of prioritized rules which together constitute the operating policy for the modeled basin and which, along with user inputs, provide the information needed to solve the simulation.

A simple guide curve rule for a reservoir is shown below. This rule looks at the Spill slot on the Hoover Dam object at the current timestep. If there is no spill, the rule reads the Guide Curve Elevation from a data slot and sets Hoover Dam's Pool Elevation equal to it. If, however, Hoover Dam is spilling, then the rule has no effect.

Example 2.1

```
HooverDam.Pool Elevation = IF (HooverDam.Spill = 0.0 [cfs]) THEN
  HooverDamData.Guide Curve Elevation[]
```

The format of the rule may look unusual if you are accustomed to programming in languages such as FORTRAN and C++. In those languages, it is normal to put the assignment statement within the logic (inside the THEN clause of the IF statement). In RPL, all slot value assignments are made at the top level of the rule. The slot being assigned a value appears on the left-hand side (LHS). The right-hand side (RHS) is a logical construct which evaluates to a single value or to nothing.

Rule Execution

When a rule executes, or *fires,* the rule is interpreted and logic of the rule is executed. Firing a rule does not imply that a slot value necessarily will be set. In the example above, if HooverDam.Spill is zero, the RHS evaluates to the Guide Curve Elevation. If, however, the spill is greater than zero, the RHS does not result in a value, and no assignment is made. It is also possible that the HooverDam.Spill slot does not have a value, in which case there is not enough information for the RHS to result in a value.

Firing of a rule only means that the rule's logical expressions will be evaluated and that the evaluation *may* result in a slot value being set. There are three possible outcomes when a rule fires.

Early termination

A rule execution which terminates early is one in which there is not enough information to evaluate all of the rule logic. This happens when a *referenced* slot (a slot on the RHS), is NaN; i.e., it does not currently have a value. If this is the case in at least one of the slot assignments, the rule terminates and no attempt is made to set any slot. This is considered a normal termination and does not stop the run or generate any errors. If you wish the rule to stop the run when a NaN slot is found, use the Stop on NaN functionality; see "Stop On NaN" in *RiverWare Policy Language (RPL)* for details.

Ineffective

An ineffective rule execution is one in which the rule logic is completely evaluated (all referenced slot values are known), but no slot values are set. This can happen for the following reasons:

- The logic determines that no slot assignment is necessary because no value is returned by the RHS.
- A value is returned by the RHS, but the slot assignment fails because the slot has already been set by a higher-priority rule. If the rule contains multiple slot assignments and if at least one assignment fails due this reason, then none of the assignments are made.

Successful

A successful rule execution is one in which at least one slot value in the model is set by the rule. To be successful requires that all the following conditions are met:

- All the information needed to evaluate all of the assignments is available; i.e., no NaNs are encountered in reference slots.

- At least one of the assignments is effective (evaluates to a number).
- None of the assignments fails due to priorities.

Rule Dependencies

A rule's dependencies are all the slots that the rule's logic referenced the last time the rule fired during the current timestep. These are the slots on which the rule's outcome depends. As a rule's logic is executed, the rule processor keeps track of all slots referenced in the logic. In practice, any slot whose value *at the current timestep* is used during a rule firing is automatically added to the dependency list for that rule.

If any of the values of the dependencies subsequently change during the rest of the timestep, the rule must *refire*. When it refires, the RHS may evaluate to a different value — resulting in a different slot assignments—and perhaps a different outcome, than during its previous firing.

Rule dependencies

A rule's dependencies is a list of dependent slots—slots that the rule referenced the last time it fired during the current timestep. These are the slots upon which the outcome of the rule execution depended. A new value in a dependent slot causes the rule to be refired.

The dependency list is generated anew each time a rule fires.

Example 2.2 Rule firing

Consider the following rule:

```
Object.SlotToBeAssigned = IF (Object.SlotA > Object.SlotB) THEN
   IF (Object.SlotA > Object.SlotC) THEN
      Object.SlotA
```

1. The first time the rule fires, the slot values are as follows:
   ```
   SlotA = 30
   SlotB = 20
   SlotC = 10
   ```
2. The rule firing starts by clearing the dependency list.
3. The first IF evaluates to true, and both SlotA and SlotB are added to the dependency list. The second IF also evaluates to true, and SlotC is added to the dependency list.
4. The rule assigns 30 to the SlotToBeAssigned, finishes successfully, and the dependency list is now: SlotA, SlotB and SlotC.
5. Now assume that during the ensuing dispatching, SlotB is recalculated to be 40, requiring that the rule fire again.
6. The second time the rule fires, the slot values are as follows:
   ```
   SlotA = 30
   SlotB = 40
   SlotC = 10
   ```
7. The rule firing starts by clearing the dependency list.
8. The first IF evaluates to false, and both SlotA and SlotB are added to the dependency list.
9. There is no ELSE clause, so there is no RHS value. The rule finished ineffectually, and the dependency list is now: SlotA and SlotB.
10. Now assume that during the ensuing dispatch, SlotC is recalculated to be 50.
11. The rule is not fired again because SlotC is not a dependency; the result of firing the rule again would not be different.
12. When a rule terminates early because one of the slots it references is a NaN, that slot is put on the rule's dependency list. This ensures that the rule will refire whenever the slot value becomes known. If the slot does not get a value during the current timestep, the rule is not put back on the agenda, and it will not refire at the current timestep.

Rules Agenda

The agenda is a list of rules eligible to be fired. The user can select whether the list is in order of priority from the highest eligible rule at the top to the lowest eligible rule at the bottom or vice versa.

Agenda

The agenda is the prioritized list of rules eligible to be fired at a given time. A rule is taken off the agenda when it fires and added back to the agenda when one of its dependencies gets a new value.

At the start of each timestep, all rules in the ruleset are added to the agenda. This ensures that each rule will be fired at least once before the end of the timestep. When the rules processor is ready to execute a rule, it fires either the highest or lowest priority rule on the agenda (depending on the selection the user has made). After a rule fires, it is removed from the agenda. Whenever any of its dependencies gets a new value, the rule is added back onto the agenda in its appropriate place according to priority. Therefore, rules always fire in the priority order specified by the user.

> **Note:** Compare the rules agenda with the Simulation dispatch queue: both are lists of items to be processed. The Simulation dispatch queue, however, processes the dispatches in the order in which they were added; whereas the rules agenda processes rules in priority order, independent of the order in which they were added to the agenda.

Example 2.3

Consider a ruleset with five rules, numbered 1 through 5. The user has selected the option to execute the agenda in descending priority order (i.e. 1, 2, 3, ...). After the start of the first timestep, three rules are executed and finish ineffectually. The next rule to be executed finishes successfully. During the ensuing dispatch, one of rule #2's dependent slots gets a new value, so rule #2 is put back on the agenda.

What rules are now on the agenda? What is the minimum number of rules which will fire before the end of the timestep? What is the maximum?

Answers:

- After the start of the timestep, the agenda is: #1, #2, #3, #4, #5.
- After three rules are executed ineffectually, the agenda is: #4, #5.
- After the next rule executes successfully, the agenda is: #5.
- After the dispatch changes a dependency of rule #2, the agenda is: #2, #5.
- At least two rules (#2 and #5) must fire before the end of the timestep.
- The maximum number of rules which will fire cannot be determined.

Rulebased Simulation Controller

The rulebased simulation *controller* orchestrates the alternation between execution of rules and solving of the simulation during a rulebased simulation run. It manages the dispatching of objects on the dispatch queue, just as the simulation controller does. When no objects can dispatch, the controller invokes the rule processor. The rule processor fires rules on the agenda, in the priority order specified by the user, until a rule is successful. Then, the controller again processes the dispatch queue.

Controller Priority

An additional function of the rulebased simulation controller is to track the controller priority when objects are dispatching. The purpose of the controller priority is to track the priority of the effects of rules as they are simulated in the model.

Controller priority

The controller priority during simulation (dispatching) is the priority of the data which is driving the current simulation. The controller priority is zero at each timestep until a rule succeeds, after which it is the priority of the last rule to succeed.

The controller priority during object dispatching reflects the priority of the information driving the dispatching. During Initialization, Beginning of Run, End of Run and each Beginning and End of Timestep, the controller priority is set to 0. During Initialization, user inputs are “set” in the slots. During Beginning of Run and Beginning of Timestep, default values are set where user input has deliberately been left out. Any slots set during these times are considered in rulebased simulation as if they had been directly input by the user. The controller priority of 0 remains in effect during the dispatching which occurs immediately after Beginning of Timestep. Since no rules have fired yet, all of these dispatches result from user input.

After a rule successfully fires and sets one or more slot values, the controller priority is set to the priority of the successful rule. As a result of these slot values, one or more objects may be ready to dispatch. Since the impending dispatch and any subsequent dispatches are a direct result of the rule’s slot assignment, these calculations are performed at the priority of the rule.

Simulation/Rules Interaction

The rulebased simulation controller behaves similarly to the simulation controller. (See “Simulation” on page 1 for details of the simulation controller.) During the Initialization, Beginning of Run and Beginning of Timestep method executions, the rulebased simulation controller behaves much like the basic simulation controller: outputs are cleared, inputs set, links are propagated, expression slots are evaluated, and objects are initialized.

This section describes the timestep specific inline rulebased simulation steps in more detail. First, the steps are presented in paragraph format, then they are presented in bullet format. During the timestep-specific inner loop of the rulebased simulation controller, control alternates between dispatching objects and the rule processor as described in the following steps:

1. Processing the object dispatch queue. Objects check their dispatch conditions and may try to dispatch (or redispatch) after a “dispatch slot” on the object changes value. An object maintains a list of its dispatch slots and dispatching only occurs if it has sufficient known values.
2. Executing rules from the agenda one at a time. When a rule is successful, all its slot assignments are made. If it is not successful, none of its assignments are made, i.e. never are some, but not all of the assignments made. When the rule succeeds, the slot cell whose value was changed is given the priority of the rule and the “R” flag. These values show up in the rules analysis dialogs. Each rule maintains a list of the slots on which the outcome of the rule is “dependent”, namely those slots that were read during the rule execution. A rule will re-execute after one or more of its dependency slots changes value. Thus, objects that are dispatching can cause rules to re-execute by changing the values of slots upon which the rules depend, while rules that change dispatch slot values can cause simulation objects to dispatch.

At the start of the timestep (after beginning of timestep behavior has occurred), the dispatch queue (step 1) is fully processed until empty. This simulates the effects of user inputs. In this processing of the dispatch queue, objects may dispatch one or more times or may not dispatch at all depending on the dispatch slots that are set or changed throughout the course of the dispatching.

Once the queue is empty, the rules controller moves to the second step and starts with the full set of enabled rules on its "agenda" in priority order. This ensures that each enabled rule gets at least one chance to execute. The controller tries to execute the first rule on the agenda, in user specified order, at which time the controller removes the rule from its agenda. A successful rule may put objects on the simulation dispatch queue. After a rule succeeds and sets a value, the controller returns to step 1 and dispatches all objects on the queue.

When the dispatch queue is again empty, the controller returns to Step 2. and executes the next rule on its agenda. Once a rule succeeds, it returns to step one and processes the dispatch queue. It continues alternating until both the dispatch queue and the rules agenda are empty.

The rulebased simulation controller follows the procedure outlined below:

1. Initialization
   - Clear all output and values set by rules from previous runs for all timesteps in all series slots.
   - Set the controller priority to 0.
   - Set user inputs
   - Propagate user inputs across links for all timesteps.
   - Determine first dispatch timestep. See "Initialization" on page 5 for details.
1. Execute rules in the Initialization Rules RPL set. See "Initialization Rules Set" in *RiverWare Policy Language (RPL)* for details.
2. Beginning of Run
   - Execute Beginning of Run methods for all objects.
   - Evaluate Beginning of Run expression slots for all timesteps.
3. For each timestep:
   a. Set the controller clock to the timestep time.
   b. Set controller priority to 0.
   c. Execute Beginning of Timestep methods for all objects.
   d. Evaluate expression slots that have evaluate-at-beginning-of-timestep selected
   e. Put all rules on the agenda (in priority order - whether 3,2,1 or 1,2,3 is user-selectable)
   f. Do the following two processes until both the dispatch queue and the rules agenda are empty
      1. Process 1. Process the Dispatch Queue: Dispatch objects (as in basic Simulation) until the queue is empty, simulating the effects of any user inputs and default values and any recent changes to slots by rules. For each slot changed by this dispatch:

         Put all rules that depend on the changed slot on the agenda, if it's not already there.

If the slot is a dispatch slot, check dispatch conditions and if necessary put the object containing the slot on the simulation dispatch queue

If the Queue is empty, move on to process 2.

2. Process 2. Execute a Rule on the agenda: Set the controller priority to the priority of the next rule on the agenda (either in order 3,2,1 or 1,2,3 based on user-selection), and fire this rule. If not successful, continue firing one rule at a time until a rule is successful and at least one slot is set in the model. Each rule is removed from the agenda after it fires. Once a rule is successful:

   Apply the slot changes, giving the changed slot cell the priority of the controller (i.e. the last rule that fired)

   Add to this rule's dependency's list each slot that was read by this rule.

   For each slot set by this rule, put all rules that depend on the changed slot on the agenda if the rule is not already there.

   For each slot changed by this rule, if the slot is a dispatch slot, check dispatch conditions and if necessary, place the object containing the slot on the simulation dispatch queue.

3. Return to Process 1 and repeat until the Agenda and the Queue are empty.

g. Set controller priority to zero, and execute End of Timestep methods on all objects.

h. Evaluate end of timestep, current timestep only expression slots

4. If the number of run cycles performed is less than the number of run cycles specified, return to Step 3. and loop through each timestep again. See "Run Cycles" on page 18 for details.

5. Execute End of Run methods on all objects including evaluating end of run expression slots.

Timestep Dependence of Rules

In RiverWare Simulation, solutions are possible forward and backwards in time. For example, reaches can solve upstream and backwards in time for some routing methods, and target operations on reservoirs solve previous timestep operations to meet a current target. In Rulebased Simulation, however, the rules should set only current values; i.e., values at the current controller timestep. The simulation should move forward in time, completely solving each timestep, then moving forward to the next.

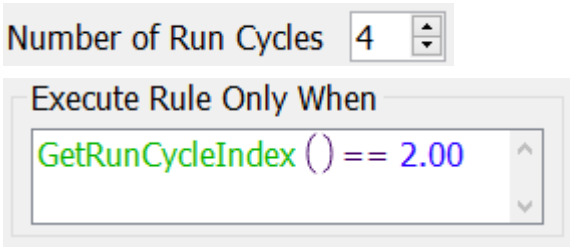
Run Cycles

Rulebased simulations can cycle through the timesteps multiple times; each pass through the run timesteps is referred to as a *Run Cycle*. Traditionally (and by default) rulebased simulations cycle through the timesteps a single time, i.e., the default number of Run Cycles is 1. You may adjust this value upwards in the Rulebased Simulation Run Parameters dialog, which is accessed from the Run Control dialog's **View** menu. See "Rulebased Simulation Run Parameters" in *User Interface* for details.

To illustrate one use of this feature, consider a model consisting of two subbasins, where the policy of the lower basin requires knowledge of the upstream basin's output for all timestep. One way to model this situation is to configure the simulation to cycle through the timesteps twice, solving for the upstream basin on the first run cycle and the downstream basin on the second. To do this, you would add execution constraints to the rules governing the

upper basin which constrain them to execute only if the current run cycle is 1, and, similarly, the remaining rules that apply to the lower basin would be constrained to execute only on the second cycle. Execution constraints and RPL expressions in general can access the current run cycle using the predefined function GetRunCycleIndex. See "GetRunCycleIndex" in *RiverWare Policy Language (RPL)* for details.

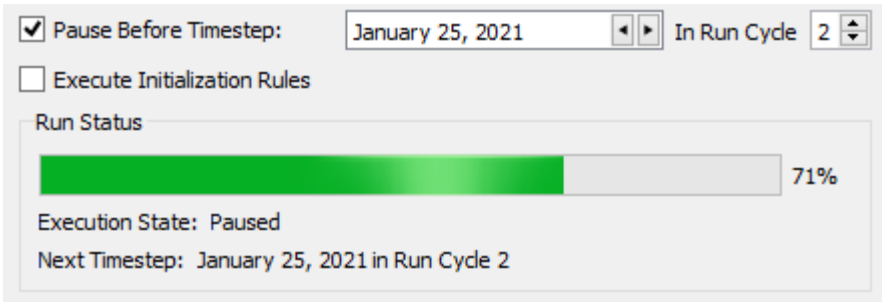
Together these features allow you to control which aspects of the policy apply to which portion of the run, which in turn allows parts of the model to solve fully before other parts—rules have access to slot values computed at all timesteps on previous run cycles.



More complex logic than that used in the previous example could allow rules/objects to solve on multiple cycles. Much like the way in which iterative MRM allows logic to control the number of runs, user logic could control the number of cycles. To do so, you would set the Number of Run Cycles parameter to a larger number, and then use logic and the STOP RUN statement to stop the run when the simulation goal had been achieved.Why might this feature by used instead of Iterative MRM? Run cycles are executed within a single run, you can run a model that is using multiple run cycles within a concurrent MRM to evaluate different hydrologies or policies.

When the number of Run Cycles to be greater than 1, the run control has additional controls for pausing as shown in Figure 2.1. Select both the timestep and the Run Cycle in which to pause. Also, notice that the Run Status at the bottom of the figure also shows the Run Cycle number.

Figure 2.1 Screenshot of Pause Before Timestep when using Run Cycles



Slot Priorities and Flags

Whenever a slot value is set in rulebased simulation, a priority is assigned to the value. This is termed the *slot priority* and can be displayed on the slot using the **View**, then **Show Priorities** menu. It indicates the priority associated with a specific value in a single timestep of a slot. The slot priority is the controller priority in effect when the slot is set. Thus, slots that are set during simulation have the priority of the last rule that succeeded. Slots that are set by rules have the priority of the rule that set them.

Slot priority

The slot priority at each timestep is the priority associated with the value in the slot. It is the priority of the controller when the slot value is set in simulation, or of the rule that set the slot value, if the slot is set directly by a rule.

In simulating the effects of a rule, the slot priorities propagate the rule priority along with the values which result from the rule's slot assignment. Slot priorities are used to determine how an object will solve and whether or not the slot value may be overwritten by values at other priorities.

User Input Flags and Priorities.

User input values are assigned the highest priority of 0. These values may never be overwritten during a run. In rulebased simulation, just as in simulation, user inputs are displayed with an I flag. Default slot values set in Beginning of Run and Beginning of Timestep also have a priority of 0, the controller priority when these methods execute. This priority is assigned because default values (usually 0.0) are filled in as a convenience where a user has deliberately chosen not to input a required value.

R Flag

Slot values set by a rule are assigned the priority of the rule which set them (an integer greater than zero). In addition to a numerical priority, slots which are directly set by a rule are displayed with a special flag, the R flag. It is indicated by the letter R next to the slot value in the Open Slot dialog. If a linked slot is set by a rule and gets an R flag, the propagated value on the other side of the link also displays the R flag.

Slots which have an R flag are more important than slots at the same priority without an R flag. This difference is discussed in more detail below.

Equivalent Slots

Some slots in RiverWare have an equivalence relationship with another slot. This means that the values of both slots are related by a function which does not change during the run. For a given value in one slot, there can be only one value in its equivalent slot. Knowing the value of one slot is tantamount to knowing the value of both slots. Pool Elevation and Storage are equivalent slots. The values of Pool Elevation and Storage are related via the Elevation Volume Table.

When a rule sets a value on a slot which is part of an equivalent slot pair, it is essentially determining the value of the equivalent slot as well. For this reason, both slots are assigned the priority of the rule. The slot whose value is being set is assigned an R flag, but the equivalent slot is not. This is because the value of the equivalent slot will not be known until it is solved for during the dispatch. The equivalent slot's value was not truly set, so it is not assigned an R flag.

When dispatching, if a value is set on a slot which is part of an equivalent slot pair, the slot is assigned the priority of the equivalent slot, regardless of the controller priority. For example, if Pool Elevation is a user input it will have a 0 priority and an I flag. If rule 1 sets Inflow, then the object can dispatch given Inflow (at a priority of 1) and Pool Elevation (at a priority of 0). All slots set as a result of dispatching will receive a priority of 1 because that is the controller priority (the priority of the last rule that successfully fired). However, the Storage slot will not receive a priority of 1. It will receive a priority of 0 because that is the priority of the Pool Elevation slot, which is the equivalent slot.

To summarize, slot priorities and flags are assigned to slot values as follows:

- 0 priority and I flag for slots set by user input.
- 0 priority for equivalent slots of slots set by user input.
- 0 priority for slots set in Beginning of Run and Beginning of Timestep (defaults).
- # priority and R flag for slots set by a rule (where # is the rule's priority).
- # priority for equivalent slots of a slot set by a rule (where # is the rule's priority).
- # priority for slots set during a dispatch (where # is the controller priority).

Resetting Slot Values

In simulation, if a slot is reset, the value must come from the same source as the previous value. (The three possible sources are user input, link propagation, or the object's dispatch method.) This is required in simulation because when objects resolve due to inter-object iterations, they must solve for the same set of known and unknowns each time. Violation results in an over-determination error.

In rulebased simulation, however, the objects need the flexibility to solve with different sets of inputs and outputs in response to new values which may be set by the rules. To accomplish this, slot values in rulebased simulation may be reset from any source.

There are, however, restrictions on the setting of a slot value in rulebased simulation. The criteria used to determine whether a slot assignment can be made are as follows:

- Values with an I flag (user input) can *never* be overwritten.
- Values that have *neither* an I flag *nor* an R flag can *always* be overwritten.
- Values that have an R flag but *not* an I flag can be overwritten depending on the new value and new priority.
  - If the new value does *not* have an R flag, the existing value can be overwritten if the priority of the new value is higher than the existing one.
  - If the new value *has* an R flag, the existing value can be overwritten if the priority of the new value is higher than or equal to the existing priority.

    Note: If the slot is part of an equivalent slot pair, the equivalent slot must also satisfy the requirement. If either slot fails the logical test, the new value is not assigned.

Table 2.1 summarizes this logic, and Table 2.2 provides examples.

Table 2.1 Rules for resetting an existing value in a slot

Existing Value Rules Flag = I	Existing Value Rules Flag = R	Slot Can be Reset	Rule Logic
Yes	No	Never	*Never* reset slot IF: • old value rules flag* = I
No	No	Always	*Always* reset slot IF: • old value rules flag* !=R

Table 2.1 Rules for resetting an existing value in a slot

Existing Value Rules Flag = I	Existing Value Rules Flag = R	Slot Can be Reset	Rule Logic
No	Yes	Depends on new value and priority	Reset slot IF: • old value rules flag* = R AND • new value rules flag !=R AND • new value priority is higher than old value priority
No	Yes	Depends on new value and priority	Reset slot IF: • old value flag* = R AND • new value flag = R AND • new value priority is higher than or equal to old value priority

Example 2.4

Table 2.2 Example slot value reset attempts

Existing Value Priority	Proposed Value Priority	Result of Attempted Assignment
3	1	Successful
1	3	Successful
6R	4	Successful
4R	6	Unsuccessful
4R	4	Unsuccessful
6R	4R	Successful
4R	6R	Unsuccessful
4R	4R	Successful
0	7	Unsuccessful
0	7R	Unsuccessful

Multi Slot Behavior

In addition to the Series Slot logic described above, Multi Slots have special logic for handling the solution of their subslots. Remember that a multislot adds a subslot for each link and that the multislot is always the total of the subslots, as in the diagram below. A single unknown, either the multislot or one of the subslots, may be solved for from the known information.

In the sample multislot in Table 2.3, the Total column could be solved for if both of the Link columns have values. Likewise, either of the Link columns could be solved for if the Total column and the other Link column have values.

Table 2.3

Total (Multislot)	Link #1 (Subslot 1)	Link #2 (Subslot 2)
NaN	15	NaN
NaN	NaN	10
25	15	10

When there is only one unknown among the multislot and the subslots, it is solved. Once all subslots and the multislot are known and one of them gets a new value, a decision must be made as to which values to retain and what value to recalculate. That decision is based on which slot was solved for the last time and the priorities of each subslot:

1. If the subslot which receives a new value is not the subslot which was solved for the last time, the multislot resolves for the same subslot as the last time.
2. If the subslot which receives a new value is the subslot which was solved for the last time, the multislot resolves for the subslot with the lowest priority.
3. If there are several subslots at the lowest priority, and these slots cannot be further prioritized by the presence of R flags, the multislot is overdetermined. In this case, an error is posted and the run stops.

In all cases where a subslot is solved for, this subslot is assigned the same priority and R flag status as the subslot which originally received a new value.

Example 2.5

Consider the following multislot, with priorities of values in parentheses.

Total (Multislot)	Link #1 (Subslot 1)	Link #2 (Subslot 2)
NaN	15 (3)	NaN

Rule #2 sets a new value on the other side of link #2. The new value propagates across the link to Subslot 2.

Total (Multislot)	Link #1 (Subslot 1)	Link #2 (Subslot 2)
NaN	15 (3)	10 (2R)

There is only one unknown (the Total column), so the multislot solves and sets the same priority on the solved-for slot as the slot which received a new value.

Total (Multislot)	Link #1 (Subslot 1)	Link #2 (Subslot 2)
25 (2R)	15 (3)	10 (2R)

If a new value were to come across either Link #1 or Link #2, the multislot would simply solve for a new value in the Total column and assign it the same priority and R flag status as the new value.

Imagine instead, that Rule #1 sets a new value on the multislot Total column.

Total (Multislot)	Link #1(Subslot 1)	Link #2 (Subslot 2)
20 (1R)	15 (3)	10 (2R)

Since the new value is on the slot which was last solved for, the multislot must solve for the lowest priority subslot. In this case, Subslot 1 would be solved for, and the priority and R flag status of the Total would be assigned to it.

Total (Multislot)	Link #1 (Subslot 1)	Link #2 (Subslot 2)
20 (1R)	10 (1R)	10 (2R)

Link Behavior

Priorities and R flags are maintained across links. Whenever a value is set on one end of a linked slot, the value, priority, and R flag (if present) are assigned to the slot on the other side of the link. Likewise, multislots propagate their priorities and R flags across their links when they solve.

Dispatching

Unlike basic simulation, an object in rulebased simulation may dispatch with different dispatch methods in a single timestep. The complication for the objects is determining which dispatch method to use whenever it gets a new slot value. Until an object dispatches for the first time during a timestep, it checks its knowns and unknowns against the dispatch conditions of its dispatch methods. This is the same process used in basic simulation. When the required knowns and unknowns are met, the object dispatches for the first time.

After the object has dispatched once at a given timestep, all of its dispatch slots should be known. If a new value is later set on a dispatch slot, the object must redispatch. But checking required knowns and unknowns is no longer a useful mechanism for determining the method with which to redispatch. In basic simulation, because slot values at a timestep are always set from the same source, the object simply redispatches with the same dispatch method used in the first dispatch.

In rulebased simulation, however, values may be set from different sources. The priorities of the dispatching slots must be used to determine the dispatch method. The dispatch method used for redispatches is the one who's required knowns have the highest priority slot values.

Governing Slots

The slots in a dispatch method's required knowns list are divided into potential *governing* slots and *non-governing* slots. Potential governing slots may determine with which dispatch method an object can solve. Non-governing slots are simply a requirement for every dispatch method. For example: Inflow and Outflow are potential governing slots, Diversion is not. Let's explore the reasons why. A river Reach object may solve upstream or downstream, depending on whether Inflow or Outflow are known. Inflow and Outflow are potential governing slots. If the object solves upstream in response to a known Outflow, the governing slot is Outflow. If, however, the object solves downstream in response to a known Inflow, the governing slot is Inflow. While both Inflow and Outflow are *potential* governing slots in this scenario, only one can be the *actual* governing slot. Diversion is a non-governing slot. A Storage Reservoir object may solve for its Inflow, Outflow, Storage, or Pool Elevation. In all cases, Diversion must be known for the Reservoir to mass balance. If Diversion is known, this only means that the Reservoir can dispatch, it tells us nothing about which of the dispatch methods the object can solve with. In general, the Governing slots are the slots which uniquely determine the dispatch method an object can solve with. The governing slots determine in which "direction" an object solves.

Potential governing slot

A potential governing slot is a slot which, by virtue of being known or unknown, will influence with which dispatch method an object can solve.

The potential governing slots for each object are listed below. All other dispatch slots are considered non-governing slots.

Table 2.4

Object	Potential Governing Slots
Aggregate Distribution Canal	Downstream Delivery Request
Aggregate Diversion Site	Total Available Water, Total Diversion Requested Total Diversion Total Depletion Total Unused Water
Aggregate Reach	Inflow, Outflow
Bifurcation	Inflow, Outflow1 Outflow2
Canal	Elevation 1, Elevation 2
Confluence	Inflow1, Inflow2, Outflow
Distribution Canal	Inflow
Diversion Object	Diversion Request, Diversion Intake Elevation
Ground Water Storage	Inflow
Inline Power Plant	Inflow, Outflow
Level Power Reservoir	Inflow, Outflow, Turbine Release, Pool Elevation, Storage, Hydrologic Inflow, Energy

Table 2.4

Object	Potential Governing Slots
Pumped Storage Reservoir	Inflow, Outflow, Pool Elevation, Storage, Energy, Pumps Used, Pump Power, PumpEnergy
Reach	Inflow, Outflow, Local Inflow
Slope Power Reservoir	Inflow, Outflow, Turbine Release, Pool Elevation, Storage, Energy
Storage Reservoir	Inflow, Outflow, Release, Pool Elevation, Storage, Hydrologic Inflow
Stream Gage	None
Water User	Incoming Available Water Diversion Requested Diversion Depletion Outgoing Available Water

Once the knowns and unknowns of an object have uniquely identified a dispatch method, the actual governing slots can be identified. The actual governing slot(s) is (are) the slots which are in the potential governing slots list and are a required known for the dispatch method. The actual governing slot(s) may be one or two, depending on the object. For Reservoir objects, there are two governing slots, selected from the list of potential governing slots. For Reach objects, there can be only one governing slot.

Governing slot

A governing slot is a slot which is:

1) in the potential governing slots list and

2) is a required known in the selected dispatch method. Governing slots provide insight into the direction in which the solution is propagating.

Example 2.6

A Storage Reservoir may solve downstream due to Inflow and Storage or Inflow and Elevation, upstream due to Outflow and Storage or Outflow and Elevation, or it may solve for its own Storage and Pool Elevation due to Inflow and Outflow. The intersection of the required knowns of the dispatch method and the potential governing slot(s) of the object, determine the actual governing slot(s).

- **Question**: What are the governing slots if a Storage Reservoir solves for its Release and Outflow?
- **Answer**: There are two dispatch methods which will result in a Storage Reservoir solving for both Release and Outflow:

  ```
  Solve given Inflow, Storage
  and Solve fiven Inflow, Pool Elevation
  ```

  The intersection of the required knowns for these dispatch method with the potential governing slots for the Storage Reservoir yields the following solutions:

  ```
  Case 1: Inflow and Storage
  Case 2: Inflow and Pool Elevation
  ```

Determination of Dispatch Method

The determination of a dispatch method for redispatching is based on the priorities of the slots. Once the object has dispatched once, all dispatch slots will be known, so checking knowns and unknowns is ineffectual. To determine the dispatch method, it is necessary to construct an alternate known slot list using the highest priority slots. RiverWare begins with an empty known slot list, then adds slots in priority order until a set of dispatch conditions is met. Since the non-governing slots are all known and they do not influence the choice of which dispatch method to solve, there is no point in prioritizing them; all non-governing slots are added to this list first. Then, the governing slots are added to the known list in priority order to determine the dispatch method.

The algorithm to determine the correct dispatch method for redispatching is as follows.

1. Begin with an empty set of known slots (pretend nothing is known).
2. Add all non-governing slots to the known slot set.
3. Add all of the object's potential governing slots with priority 0 to the known slot set.
   a. Check the known set against the required knowns of each dispatch method. If a match is found, put the object on the queue with this dispatch method, and exit the algorithm.
   b. Check the known set against the conflict list.If the known set contains all of the slots of a conflict, put the object on the queue with the same dispatch method as the last time it dispatched, and exit the algorithm.

4. Loop through all priority levels starting with the highest priority (1,2,3...) until a dispatch method is determined. For each priority level:
   a. If there are any potential governing slots at the priority level that have an R flag, add the R-flagged slots to the known set and defer the other slots.
   b. If no potential governing slots at the priority level have an R flag, add all potential governing slots at that priority to the known set.
   c. Check the known set against the required knowns of each dispatch method. If a match is found, put the object on the queue with this dispatch method and exit the algorithm.
   d. Check the known set against the conflict list. If the known set contains all the slots of a conflict, put the object on the queue with the same dispatch method as the last time it dispatched and exit the algorithm.
5. Loop through all priority levels again in order (1,2,3...), and for each:
   a. Add any non-R-flagged slots that were deferred in Step 4. to the known set.
   b. Check the known set against the required knowns of each dispatch method. If a match is found, put the object on the queue with this dispatch method and exit the algorithm.
   c. Check the known set against the conflict list. If the known set contains all the slots of a conflict, put the object on the queue with the same dispatch method as the last time it dispatched and exit the loop.

This algorithm always results in the identification of a dispatch method or an error and terminated run.

Example 2.7

A Level Power Reservoir, named ResA, has no diversions and a user input Hydrologic Inflow. The following sequence of events take place.

1. A low priority rule (#4) sets an Outflow on a reservoir directly upstream of the ResA. The value and the R flag propagate across the link to this ResA's Inflow slot.
2. A higher priority rule (#3) sets the ResA's Storage.
3. ResA can now dispatch for the first time with the Solve given Inflow, Storage dispatch method at a priority of 3.

After the dispatch, all of the reservoir's dispatch slots are known at the following priorities.

Slot	Priority
Inflow	4R
Hydrologic Inflow	0
Storage	3R
Pool Elevation	3
Diversion	0
Return Flow	0
Outflow	3

Now suppose that the highest priority flood control rule (#1) has to overwrite the ResA's Outflow to prevent flooding downstream. The reservoir must redispatch.

- **Question**: What slots does the ResA's known set contain after each step of the logic to determine the dispatch method? Do any of these known sets match a dispatch method? Which one?

  Answer: The logic is as follows:

  1. Start with an empty known set.
  2. Add all of the non-governing slots to the known set.

Slot	Priority
Diversion	0
Return Flow	0

  3. Hydrologic Inflow (a potential governing slot) was user input and has a priority of 0. Add this slot to the known set.

Slot	Priority
Diversion	0
Return Flow	0
Hydrologic Inflow	0

     These three slots alone do not satisfy any of the dispatch conditions. Continue.

     These three slots alone are not a conflict. Continue.

  4. Loop through the priorities:
     - Priority 1: Outflow is known at priority 1R. Add it to the known set. Hydrologic Inflow, Diversion, Return Flow, and Outflow do not satisfy any dispatch conditions. Hydrologic Inflow, Diversion, Return Flow, and Outflow are not a conflict. Continue.

Slot	Priority
Diversion	0
Return Flow	0
Hydrologic Inflow	0
Outflow	1R

     - Priority 2: There are no slots at this priority.

       No changes are made to the known set. Continue.

Slot	Priority
Diversion	0
Return Flow	0
Hydrologic Inflow	0
Outflow	1R

     - Priority 3:

Both Storage and Pool Elevation are at priority 3, but Storage has an R flag. Add Storage to the known set and defer Pool Elevation.

Diversion, Return Flow, Hydrologic Inflow, Outflow, and Storage satisfy the conditions for the Solve given Outflow, Storage dispatch method. Add this object to the queue with this dispatch method, and exit.

Slot	Priority
Diversion	0
Return Flow	0
Hydrologic Inflow	0
Outflow	1R
Storage	3R

Note: The Pool Elevation and Inflow slots are never added to the known set. It is not necessary to consider them because we satisfied a dispatch method's conditions with higher priority slots and R flags.

- **Question**: Once the object redispatches, a new value will be computed for two of the slots. Which slots? What will their priorities be after the dispatch completes?

  Answer: The object will redispatch using the solveMB_givenOutflowStorage dispatch method at a controller priority of 1 (because the last rule to succeed was rule #1). During the dispatch, the object will solve for new Inflow and Pool Elevation values. These values will be set on the Inflow and Pool Elevation slots at a priority of 1 and 3, respectively. (The Pool Elevation slot is an equivalent slot to Storage which has a priority of 3R.)

Chapter 3
Multiple Run Management

Multiple Run Management (MRM) is a RiverWare utility for setting up and automatically running many runs. The runs are set up through the Multiple Run Manager, which carries out all the runs and then outputs the results to a RiverWare Data Format (RDF) file, Excel file, or output DMI.

Topics

- About Multiple Runs, page 31
- Running a Preconfigured MRM Model, page 31
- Managing MRM Configurations, page 32
- Setting Up a Multiple Run Configuration, page 34
- Saving and Restoring Initial Model State, page 59
- Distributed Concurrent Runs, page 60
- RDF Viewer, page 69

About Multiple Runs

A multiple run is defined as a set of one or more model runs. For all runs:

- The model configuration (object network) remains constant.
- The timestep size is constant.
- The same set of slots is saved to the output.

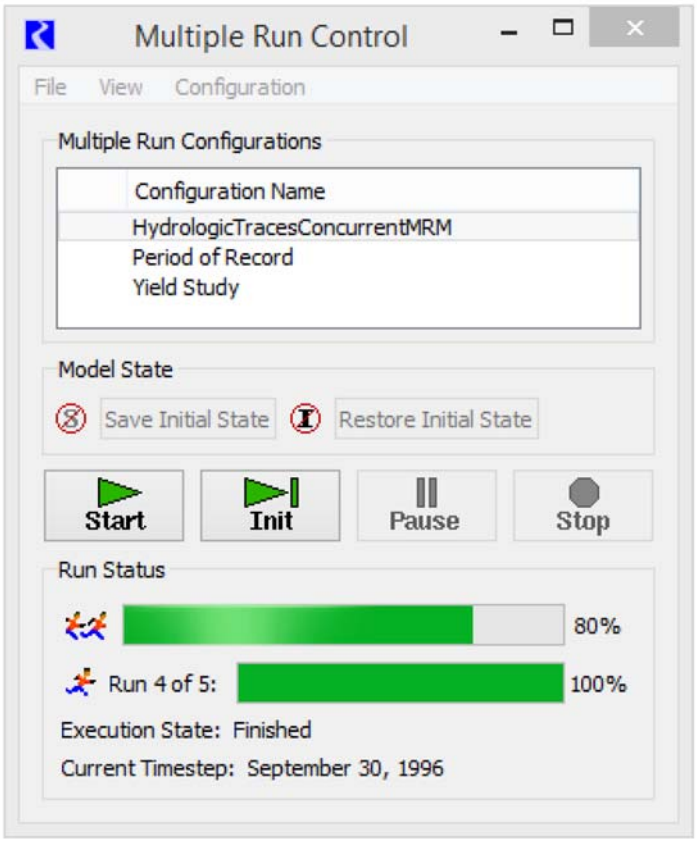
Runs are conducted automatically; that is, RiverWare controls and invokes each run without user interaction.

Running a Preconfigured MRM Model

This section describes how to run a preconfigured MRM model. It is assumed that the MRM configuration is already set up and you wish to run the multiple runs and view the multiple run output.

- Open the MRM Dialog by selecting **Control**, then **MRM Control Panel** from the main RiverWare menu bar or select **MRM**  on the toolbar.

- Highlight the desired configuration.
- Press the **Start** button.



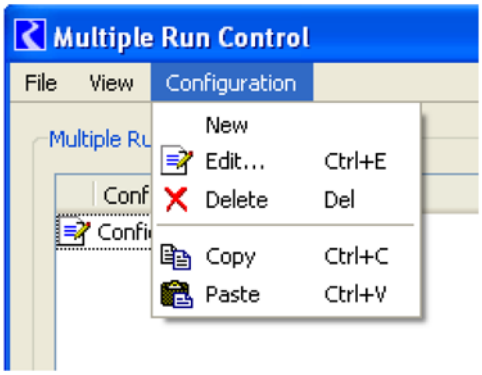
- After the run has finished, determine where the output was placed by selecting **Configuration**, then **Edit** and selecting the **Output** tab in the ensuing Multiple Run Editor dialog.
  - If there is a value in the Control File field this means that data was sent to the RDF file (or files) specified in the control file using the "file=" keyword. If there are no "file=" keywords specified in the control file, the data will go to the file specified in the Data File field. If **Generate Excel files from RDF files** is selected, an Excel spreadsheet was also created in the same folder and by the same name as the control file.
  - If there is a value in the DMI field, this means that data was sent to a database using a DMI. Check the diagnostics output window for DMI diagnostics that show where output was sent. DMI slot diagnostics must be turned on during the run for this information to be printed.

See "Output" on page 52 for details. See "RDF Viewer" on page 69 for more information on quickly viewing the results of an MRM run using the RDF Viewer.

Managing MRM Configurations

This section describes how to manage MRM configurations using the Multiple Run Control dialog. The basics of creating, deleting, and opening a configuration for editing are described here.

- Open the MRM Dialog by selecting **Control**, then **MRM Control Panel** from the main RiverWare menu bar or select the **MRM** button on the toolbar. 



Creating a New Configuration

- Select **Configuration**, then **New** from the Multiple Run Control Dialog.
- Double-click the newly created configuration (or highlight it and select **Configuration**, then **Edit** or right-click the highlighted configuration to bring up a context menu and select **Edit**).
- In the MRM Configuration, make any necessary edits to the new configuration and select **OK**. See "Setting Up a Multiple Run Configuration" on page 34for details.

Editing an Existing Configuration

- Highlight the desired configuration in the Multiple Run Control dialog and select **Configuration**, then **Edit** or right-click the highlighted configuration to bring up a context menu and select **Edit**.
- Apply desired edits in the MRM Configuration dialog and select **OK**.

Deleting an Existing Configuration

- Highlight the configuration to be deleted and select **Configuration**, then **Delete** or right-click the highlighted configuration to bring up a context menu and select **Delete**.
- Apply the deletion by confirming the dialog

Copying an Existing Configuration

An easy way to create a new configuration similar to an existing one is by copying the existing configuration and making changes. Configurations can only be copy and pasted within one model (i.e., within one open session of RiverWare).

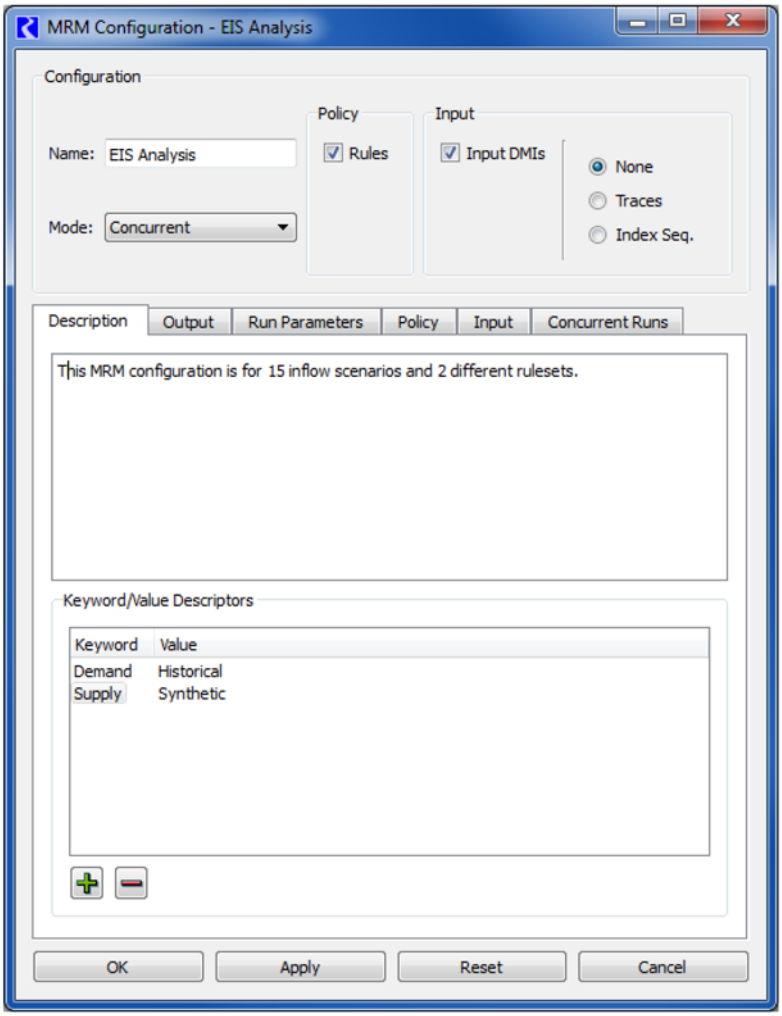
- Highlight the configuration to be copied and select **Configuration**, then **Copy** or right-click the highlighted configuration to bring up a context menu and select **Copy**.
- Select **Configuration**, then **Paste** to paste the copied configuration into the open Multiple Run Control dialog or right-mouse select the highlighted configuration to bring up a context menu and select **Paste**.
- Highlight the "Copy of" configuration and select **Configuration**, then **Edit** (or right-click the highlighted configuration to bring up a context menu and select **Edit**).
- Change the name, if desired, and make any other edits. Select **OK** in the Multiple Run Editor.
- Edit notations. Whenever a configuration is edited, the change is noted with an icon in the Multiple Run Control dialog until those changes have been either accepted or canceled. 

Setting Up a Multiple Run Configuration

This section provides the steps to setting up a multiple run configuration. Setting up a multiple run configuration requires specifying several parameters in the MRM Configuration. The parameters and brief descriptions are provided below.

- Open the Multiple Run Control dialog by selecting **Control**, then **MRM Control Panel** from the main RiverWare menu bar or select the **MRM** button on the toolbar. 

- Create a new configuration and edit an existing configuration, as appropriate. See "Creating a New Configuration" on page 33 and "Editing an Existing Configuration" on page 33 for details.



Name

Provide a unique name for the configuration in the **Name** field.

Description

The configuration description in the upper panel may be multiple lines of text. This first non-blank line of the description appears in an MRM output RDF file. This text is optional, and will not affect the MRM execution if left blank.

Keyword/Value Descriptors are user-entered to describe the MRM configuration. For example, these could indicate something about the data being brought in by DMIs, something about the ruleset, or anything else the user would

like to document. These descriptors can optionally be written into CSV or netCDF files output from the multiple run as described. See “CSV Files” on page 54 and “NetCDF Files” on page 57 for details.

Mode

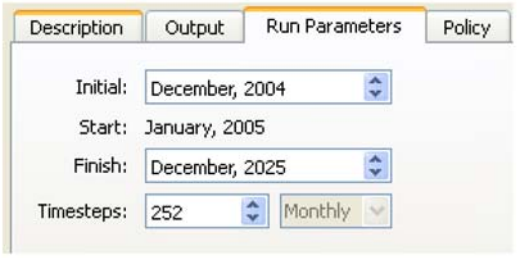
Select the Mode of the configuration (Concurrent, Consecutive or Iterative) from the **Mode** menu. Selecting a mode will add the appropriate tabs to the dialog. Configuration of these tabs is described for each of the modes.

Concurrent Runs

Concurrent runs are multiple runs of which the time horizons are identical. The begin and end times, and timestep length, are the same for all runs. Concurrent runs are used to run the model multiple times with different inputs for each run. Inputs include rulesets (policy alternatives), input DMIs (i.e. series data like alternative hydrologies), and/or index sequential (data sampling technique).

- For concurrent mode, the **Run Parameters** tab is shown. The **Run Parameters** tab describe the initial and end date of each concurrent run. The run parameters also show the timestep size but timestep size is dependent on the model and can only be changed in the single run control dialog.

  Note: The Initial and Finish timestep for a concurrent run can be different than what is shown in the single Run Control.



- The **Concurrent Runs** tab shows the number of runs that will be made.
  - In most cases, the total number of runs should be the product of the number of rulesets, the number of input DMIs, and the number of index sequential runs:

    ```
    #of MRM runs = #of Rulesets * #of Input DMIs * #of Index Seq Runs
    ```

  - If the Pairs option for DMI/Index Sequential Mode has been selected on the **Input** tab, the total number of runs should equal the number of pairs of Input DMIs and Index Sequential runs. If the number of input DMIs does not match the number of Index Sequential runs, then the number of index sequential runs equals the total number of possible pairs, (i.e., the minimum of the number of input DMIs and the number of Index Sequential runs). (See the Index Sequential / DMI Mode section for more details.)

    ```
    #of MRM runs = min(#of Input DMIs, #of Index Seq Runs)
    ```

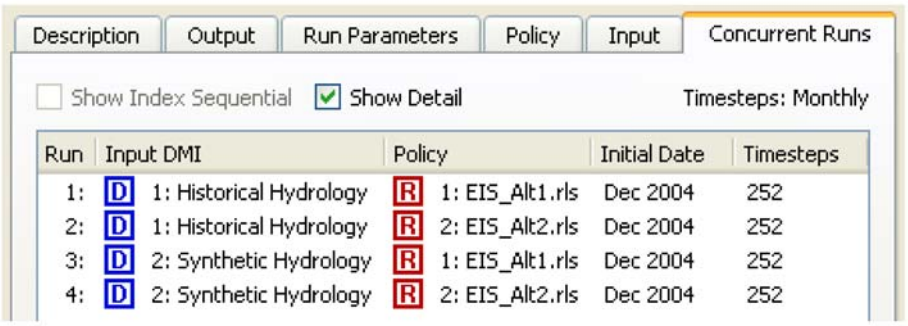
Figure 3.1 shows Concurrent Runs with 2 Input DMIs and 2 policy sets, no Index Sequential.

Figure 3.1



As shown in Figure 3.2, selecting the **Show Details** checkbox will show the names of the DMIs and policy sets.

Figure 3.2

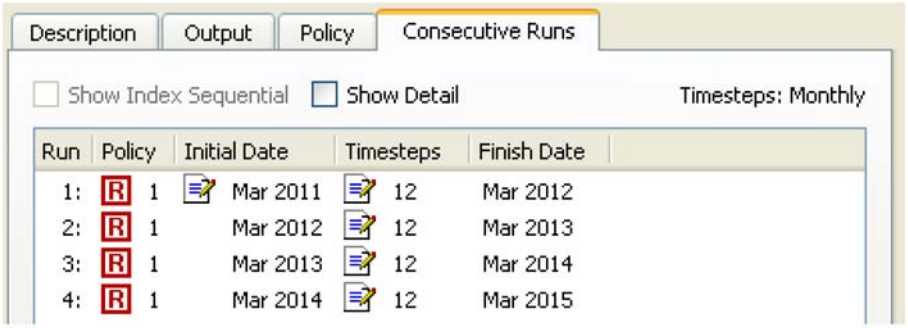


Consecutive Runs

Consecutive runs are multiple runs where the time horizons are laid out consecutively. The end time of one run is the initial time of the next run. Timesteps do not vary among the different runs in a consecutive run.

- On the **Consecutive Runs** tab, the **Edit** button indicates which fields in this dialog are editable. If necessary, change the **Initial Date** for the consecutive runs. Do this by selecting the existing initial date and toggling the up/down arrows in the ensuing date-time spinner.
- Change the number of timesteps for the first consecutive run by selecting the existing number of timesteps and toggling the up/down arrows of the ensuing integer spinner. The Finish Date automatically updates.
- Append a new row for each desired additional consecutive run by selecting the **Plus** button. Or right-click below the existing consecutive run and selecting **Append Row** from the ensuing context menu. The Initial Date is automatically set to match the Finish Date of the previous consecutive run. In the Initial Date column, only the first consecutive run can be changed.
- Change the number of timesteps for each individual run, if necessary. The default is for newly appended runs to have the same number of timesteps as the previous run.

- To remove a run, select the **Minus** button. This will always remove the last run.



Iterative Runs

Iterative runs are multiple runs where MRM rules at the beginning and/or end of each run examine the state of the system and, if appropriate, set values for the subsequent simulation run. If no values are set or the maximum number of iterations occurs, then the simulation ends. As in concurrent runs, the time horizons, begin and end times and timestep length are all the same for all runs.

The iterative runs can use any of the controllers as specified in the single Run Control dialog: simulation (with or without accounting), rulebased (with or without accounting) or optimization. If the run is rulebased or optimization, the same RPL set or Goal set, respectively, is used in each iteration.

An iterative run executes as follows:

1. Initialize the iteration count.
2. Execute the Pre-MRM Run Rules, if specified.

   Note: Pre-MRM Run Rules are similar to Initialization rules for each run; see "Initialization Rules Set" in *RiverWare Policy Language (RPL)*.
3. Perform a single run.
4. Execute the Post-Run Rules, if specified.
5. If the Post-Run Rules return "no change", that is they do not assign one or more new (different) values, the iteration is complete.
6. Otherwise, the iteration count is checked. If it equals the maximum number of iterations specified, then the iteration is complete also.
7. If the iteration is not complete, then increment the iteration count and return to Step 3..

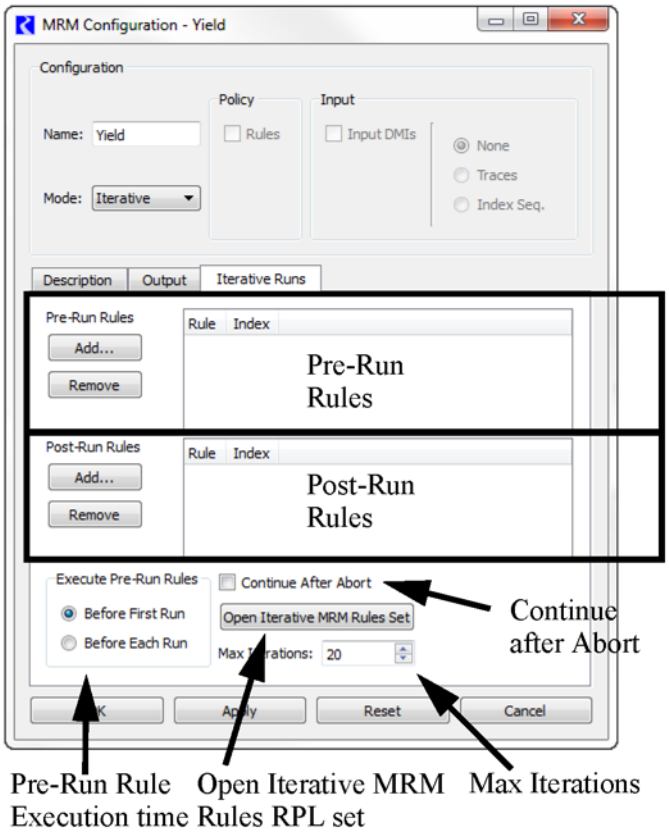
When an MRM rule, either pre-run or post-run, sets a value on a slot, the value is given the **i** flag indicating that it is a "Iterative MRM" flag. Values with the iterative MRM flag are cleared at the beginning of an iterative MRM run but are not cleared between iterations of the MRM. This allows values set by the iterative MRM rules to persist between iterations but values are cleared at the beginning of the MRM run. In non-iterative MRM and single runs, values with an **i** flag are also cleared. You should be aware of this behavior if switching from iterative MRM to another mode.

Values set by the Iterative MRM "i" flag behave with output semantics. That is, they can be overwritten by any other value. In rulebased simulation, they are set when the controller is at priority zero, so values are given a

priority of zero. Iterative MRM rules should only set values on data objects (typically integer indexed series slots as described at the end of this section), not simulation objects. Iterative MRM rules should be used to control the multiple runs; rulebased simulation rules within the run should be then used to set values on simulation slots that will actually be used in the run.

To configure an iterative run:

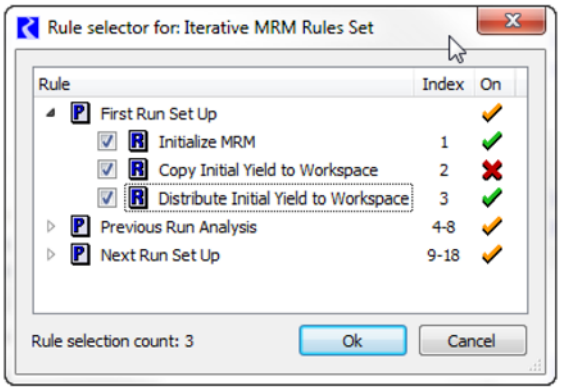
- Within the MRM Configuration dialog with the iterative mode, there is no **Run Parameters** tab. The iterative runs will begin at the start timestep of the model run. If users wish to change the model start timestep, this should be done in the single Run Control dialog.
- On the **Iterative Runs** tab are the following significant areas: the Pre-MRM Run Rules, the Post-Run Rules, the **Continue After Abort** toggle, **Pre-Run Rule Execution Time**, the **Open Iterative MRM Rules Set** button, and the **Max Iterations** selector.



- Select **Open Iterative MRM Rules Set** button to open the Iterative MRM Rule set editor. This dialog operates very similarly to the standard RBS Ruleset Editor dialog. Create one (or more) Policy Group(s) and associated rules. The MRM Rules created are stored within the model file and may be available for use with any number of iterative MRM configurations. This set of rules can also be accessed from the workspace **Policy**, then **Iterative MRM Rules Set**.
- Once the MRM Rules have been built, return to the **Iterative Runs** tab of the MRM Configuration dialog. In this tab, the define which of the MRM rules should execute as part of the PreRun rules and which should execute after each iterative run, or both. These are defined in the appropriate areas using one of the following methods

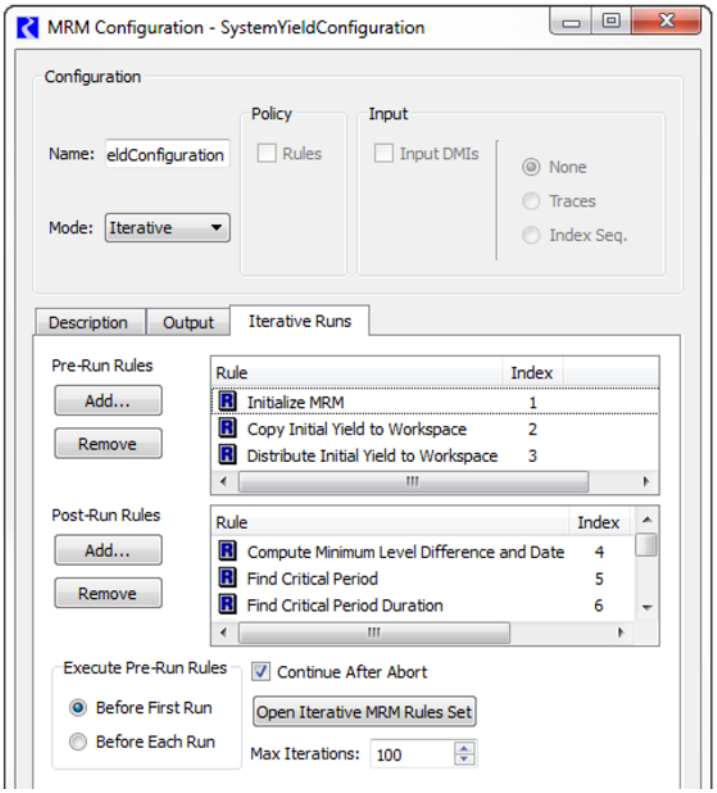
  - Use the **Add** and **Remove** buttons. Select the **Add** button to bring up the Rule Selector as shown in the following figure. This dialog shows the rule groups in a tree view. Expand the tree-view to show individual rule names, their Index and their On status. Select the checkbox to check the desired rules. Select **Ok** to accept and return to the configuration dialog.
  - Drag and drop rules from the MRM Ruleset to the Pre-Run Rules area or Post-Run Rules area.

    Note: The order of addition into the display does not affect the rule ordering; the order is strictly consistent with priorities defined in the MRM Rules.



- To view a rule directly from the **Iterative Runs** tab, double-click the rule's name to open the Rule Editor.
- Use the **Execute Pre-Run Rules** buttons to specify when Pre-Run Rules are executed:
  - **Before First Run**. This is the default. Choose this option to execute the Pre-Run Rules before the first single run only, but not before subsequent runs.
  - **Before Each Run**. Choose this option to execute the Pre-Run Rules before *each* single run.
- Consider activating the **Continue After Abort** option. If you want the Post-Run Rules to execute and possibly the next iteration to begin after an iteration is prematurely aborted (for any reason), select the checkbox.
- Select the **Max Iterations** selector and adjust it using the up and down arrows or by typing a value in the field to define the maximum number of iterations to run. Figure 3.3 shows a fully configured **Iterative Runs** tab.

Figure 3.3



Note: The Post-Run Rules must make a significant change in values (for example, through a rule assignment or import of data through an input DMI) at the end of each run for the controller to make the next iteration. If the Post-Run Rules do not set any values, (possibly due to convergence) the controller will assume that the goal of the iterations has been achieved and stop.

Note: The MRM rules execute differently from the basic ruleset. When the MRM rules execute, each fires once and only once according to the execution order specified. There is no dependency functionality. In fact, they will all fire even if one of them aborts.

Integer indexed Series Slots work very well with Iterative MRM. Using these slots, you can store inputs, outputs, or intermediate results based on the index of the run. For example, after each iterative run, you could store the total volume of water released in an integer index slot in the row corresponding to the run index. At the end of the entire run, all of these volumes are stored and can be reviewed. The GetRunIndex predefined function can be used to get the index of the next iterative run. See "Integer-indexed Series and Agg Series Slots" in *User Interface* and "GetRunIndex" in *RiverWare Policy Language (RPL)* for details.

Policy

For concurrent and consecutive mode, rulesets can be specified for the runs. The policy setup of the configuration (None or Rules) must be selected from the Policy section of the MRM Configuration. Selecting **Rules** will enable the Rulebased Simulation controller when the multiple run is started. Rulebased multiple runs are runs in which there are one or more rulesets. You specify the ruleset to use for each run. Variations in rules can be a function of differences in content, priorities, or both.

- Select **Rules** under in the Policy section of the MRM Configuration and select the **Policy** tab.
- Append a new row for each additional ruleset by selecting the **Plus** button  at the bottom.
- Select the ruleset by selecting the **File Chooser** button  or double-click and type in the path name of the ruleset directly. This allows you to use environment variables in the path. Environment variables are prefixed with the "$".
- If necessary, remove a ruleset by selecting the **Delete** button  . There is no confirmation, but you can restore ruleset rows using the **Reset** button.

In Figure 3.4, the first ruleset was specified by browsing to the C: drive, the second ruleset was specified by typing the path in directly and using the environment variable EISDIR

Figure 3.4



Note: The **Policy** tab is not applicable to Iterative MRM runs, but an iterative run can be a rulebased run where one ruleset is used for all iterations. It must be explicitly opened and loaded before the iterative run is started.

Note: Multiple rulesets in a single MRM configuration are not supported if using Distributed Multiple Runs; see "Distributed Concurrent Runs" on page 60 for details.

Input

The inputs to the multiple runs are specified by selecting options in the top of the dialog and then specifying configuration details in the appropriate tabs.

- If the Multiple Runs use an Input DMI, select the **Input DMIs** checkbox in the Input section.
- Optionally specify one **Initialization DMI**; see "Initialization DMI" on page 43.
- Specify whether to use **Traces**, **Index Seq.**, or **None**. See "Index-sequential Runs" on page 44 for details on Index Sequential. See "Input DMI Runs" on page 43 for details on traces.

Note: The Input section and tab are not applicable or available to Iterative runs.

Initialization DMI

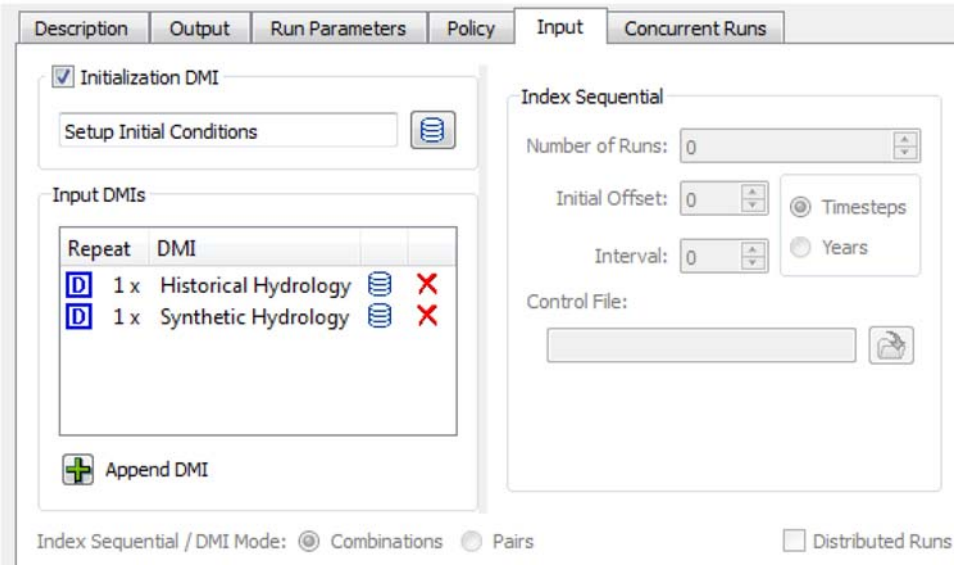
You can optionally specify a single DMI or group that is invoked at the beginning of the multiple run. Select the **Initialization DMI** toggle to show the entry field. Type in the name of an existing DMI or use the button to select an the DMI or group.



Input DMI Runs

Input DMI runs vary the input data for a specified set of slots. For instance, ten runs might represent ten alternative release schedules for a reservoir. You specify the number of runs and a DMI for each run. The DMI loads the data into the model for each run. The DMIs must be previously defined in the RiverWare DMI interface.

- When the Input DMIs is selected on the MRM Configuration, on the **Input** tab, the Input DMIs section becomes active.

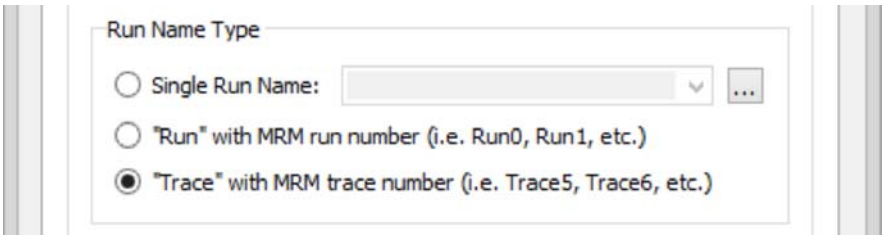


- Append a new DMI by selecting the **Plus** button at the bottom
- Select the input DMI by selecting the **DMI** icon and then choosing the previously configured DMI from the list.
- If the DMI will be called more than one time, change the number in the Repeat column by double-clicking the cell and using the up/down arrows. DMIs might need to be called more than once in a MRM run, for example, with runs that execute multiple traces using the same DMI. In this situation, the DMI needs to be configured to make use of the trace number.

  - For a Control File-Executable DMI, the executable should reference the last command line parameter passed from RiverWare:

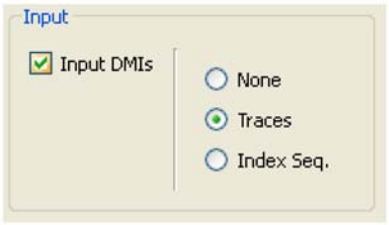
    ```
    -STrace=traceNumber
    ```

  - For an Excel DMI, the Excel Dataset Run Name Type setting should use either the **Run** or **Trace** option; see "Run Name Type" in *Data Management Interface (DMI)*.



- If you are using Input DMIs but not using Index Sequential, you can choose the **Traces** option from the Input section on the MRM Configuration. This allows you to run only a portion of the runs specified by the Input DMIs. When Traces are specified, the Input tab shows the Traces area. In this area, you can specify
  - **First Trace**
  - **Number of Traces**
  - To not run a subset of the runs, select **None**.





- If you wish to use Ensembles with HDB datasets, select the **Input Ensembles** checkbox. See "Ensembles" on page 47 for details.

Index-sequential Runs

Index Sequential runs use an input time series which is systematically shifted between runs. You specify the number of runs, the interval (number of timesteps) by which the input data is shifted from one run to the next, and an initial offset (number of timesteps) by which data is shifted for the first run. The slots with input data to be shifted are identified by in a DMI control file. This type of run is typically used to perturb (shift) historical hydrologic data in order to be able to conduct statistical analysis of the results.

Note: Index Sequential is not applicable to Iterative or Consecutive runs.

Index Sequential specifies that, given a run start time, run timestep, run duration, number of runs, and time offset, input time series, a run can be systematically perturbed between runs. For example, if an Index Sequential run is defined as the following:

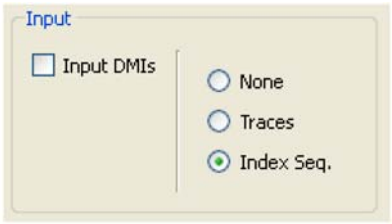
- input time-series:
- original value vector: 1, 2, 3, 4, 5, 6, 7, 8, 9
- original time vector: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep
- offset: 4 timesteps
- interval: 1 timestep
- number of runs = 3

then, the following sequence of runs occurs:

- Run 1: Jan = 5, Feb = 6, Mar = 7, Apr = 8, May = 9, Jun = 1, Jul = 2, Aug = 3, Sep = 4
- Run 2: Jan = 6, Feb = 7, Mar = 8, Apr = 9, May = 1, Jun = 2, Jul = 3, Aug = 4, Sep = 5
- Run 3: Jan = 7, Feb = 8, Mar = 9, Apr = 1, May = 2, Jun = 3, Jul = 4, Aug = 5, Sep = 6

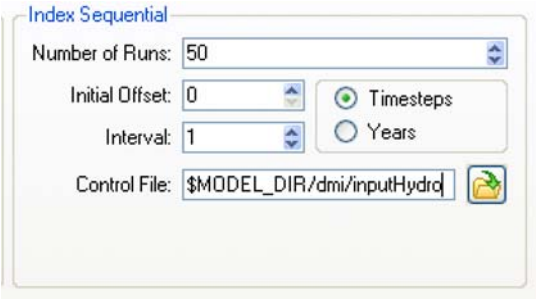
To setup an Index-sequential run:

- Select **Index Sequential** in the Input section of the MRM Configuration and select the **Input** tab.

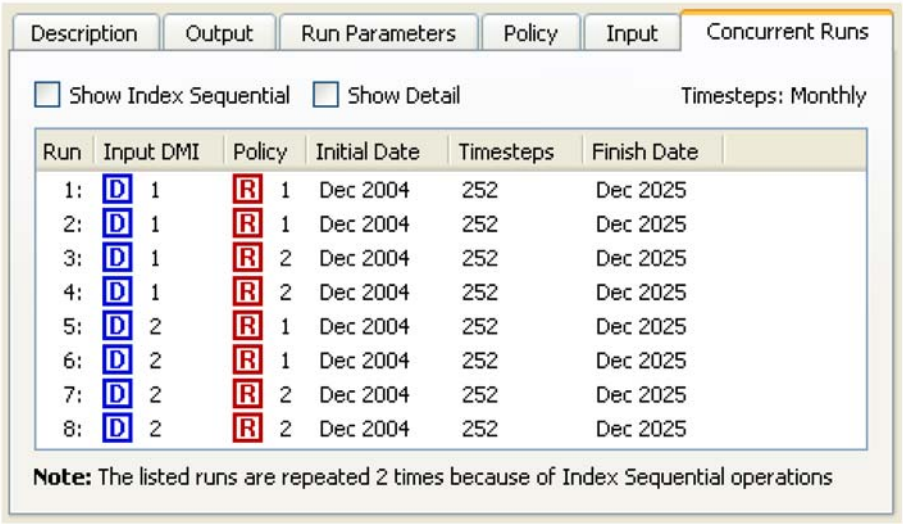


- Specify the **Number of Runs** by typing in the value or by toggling the up/down arrows.
- Specify the **Initial Offset** by typing in the value or by toggling the up/down arrows.
- Specify the **Interval** by which the data is shifted from one run to the next by typing in the value or by toggling the up/down arrows in the Interval field.
- Specify the units for the Initial Offset and Interval as **Timesteps** or **Years**.
- Specify the **Control File** by typing in the path directly or by using the file chooser to select it. Typing in the path and name directly allows for the use of environment variables. Take care to spell the entire path correctly. The control file specifies the slots that will be rotated for each run. It uses the same format specified, but no file

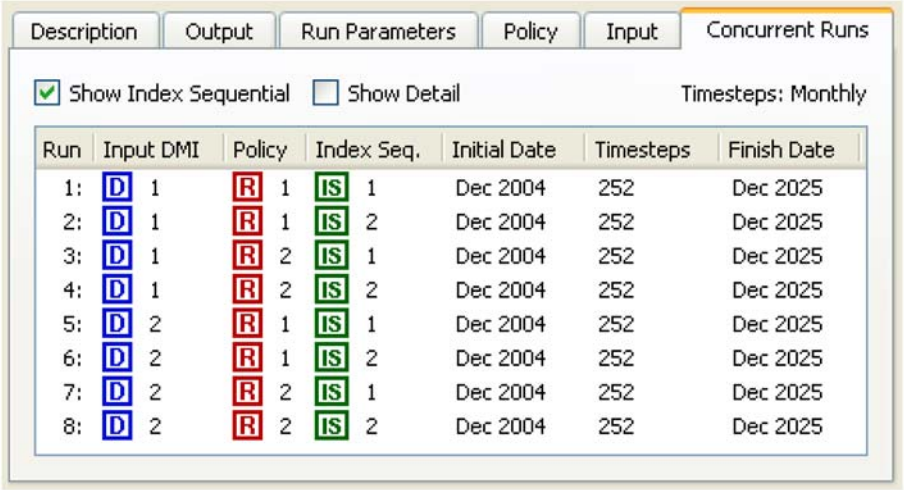
name or keyword value pairs are necessary. Typically, it is just the object and the slots that are specified. See "Control File" in *Data Management Interface (DMI)* for details.



- On the **Concurrent Runs** tab, the combinations of policy sets and Input DMIs are set up for the index sequential run.



- Select the **Show Index Sequential** checkbox at the top of the **Concurrent Runs** tab to view each index sequential run listed individually.



Combined Runs

By default, the runs that are made in concurrent MRM are a multiplicative combination of inputs. This set is called *Combined Runs*. Combined runs are defined as the Cartesian product of all the involved base-type runs. For example, a combined run consisting of two rulesets and four Input DMIs, results in 2 * 4 = 8 model runs.

The order in which individual runs within a combined multiple run are executed is determined by the precedence level of each mode. Order of precedence (from highest (1) to lowest (3)) is as follows:

1. Input DMI runs,
2. Rulebased runs, and
3. Index-Sequential

Elements of lower precedence iterate before elements of higher precedence. For example, in case of a combined run containing two Input DMI runs and two Rulebased runs, the following sequence of four runs is made:

1. Input DMI 1 & Ruleset 1.
2. Input DMI 1 & Ruleset 2.
3. Input DMI 2 & Ruleset 1.
4. Input DMI 2 & Ruleset 2.

Thus, the default combinations mode results in the total number of runs equal to the product of the number of policy sets, the number of input DMIs, and the number of index sequential runs:

#of MRM runs = #of Policy Sets * #of Input DMIs * #of Index Seq Runs

In very specific circumstances, it is possible to alter the mode of how many MRM runs result from the combination of input DMIs, policy sets, and Index Sequential runs. If Index Sequential has been selected *and* there are as many (or more) Input DMIs as Index Sequential runs *and* there are zero or one rulesets selected, then it is possible to choose either **Combinations** or **Pairs** in the **Index Sequential / DMI Mode** selection.

The Pairs mode results in the total number of runs equaling the number of pairs of Input DMIs and Index Sequential runs. If the number of input DMIs does not match the number of Index Sequential runs, then the number of index sequential runs equals the total number of possible pairs, (i.e., the minimum of the number of input DMIs and the number of Index Sequential runs). The Pairs mode is necessary to run the CRSS Lite model.

# of MRM runs = min(# of Input DMIs, # of Index Seq Runs)

When the pairs mode is specified, the runs can be distributed to multiple processors. See "Distributed Concurrent Runs" on page 60 for details.

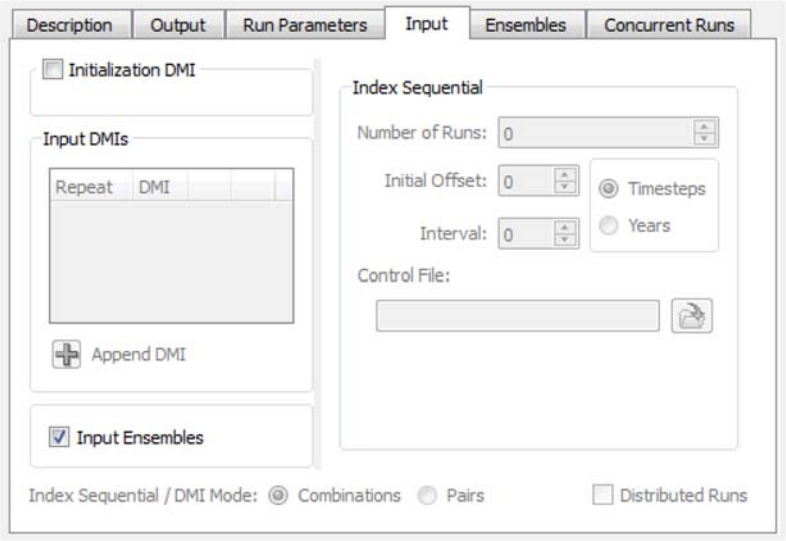
Ensembles

An ensemble contains a set of traces where each trace represents the data for one run of the multiple run. An ensemble of trace data can function as input to the runs of a multiple run, or can represent output from a multiple run. An ensemble can have metadata that describes the overall ensemble and metadata that describes each of the individual traces.

Note: Currently only HDB datasets in Database DMIs can be configured to function as ensembles.

Ensemble Configuration

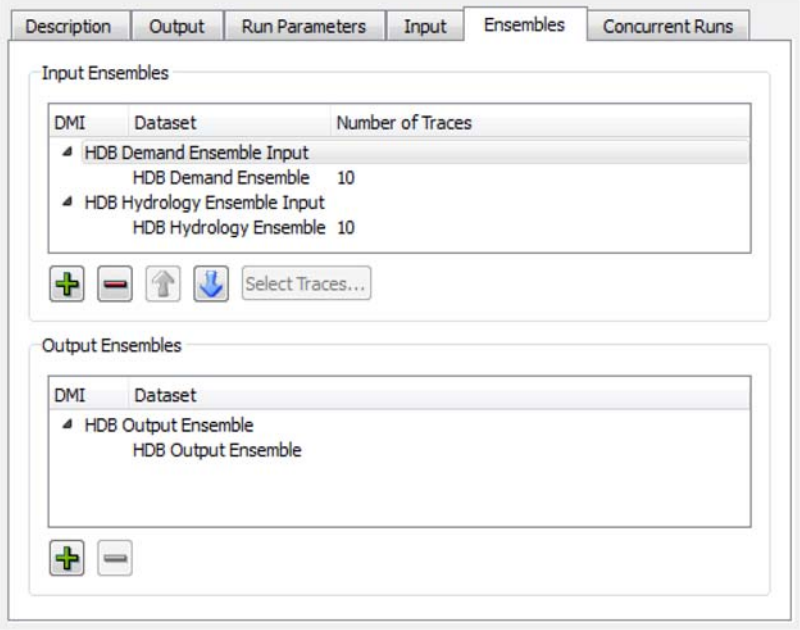
To utilize ensembles, on the **Input** tab of the MRM configuration, select the **Input Ensemble** checkbox. This will disable Input DMIs, Traces, and Index Sequential functionality as the ensembles will determine the number of runs in the multiple run.



When you select the **Input Ensembles** toggle, an **Ensemble** tab is added to the dialog where input and output ensembles are specified.

Select the **Plus** in the **Input Ensembles** frame to open a list of input DMIs defined in the model. You can select one to add as an ensemble. Only valid input DMIs with datasets configured with ensembles can be added.

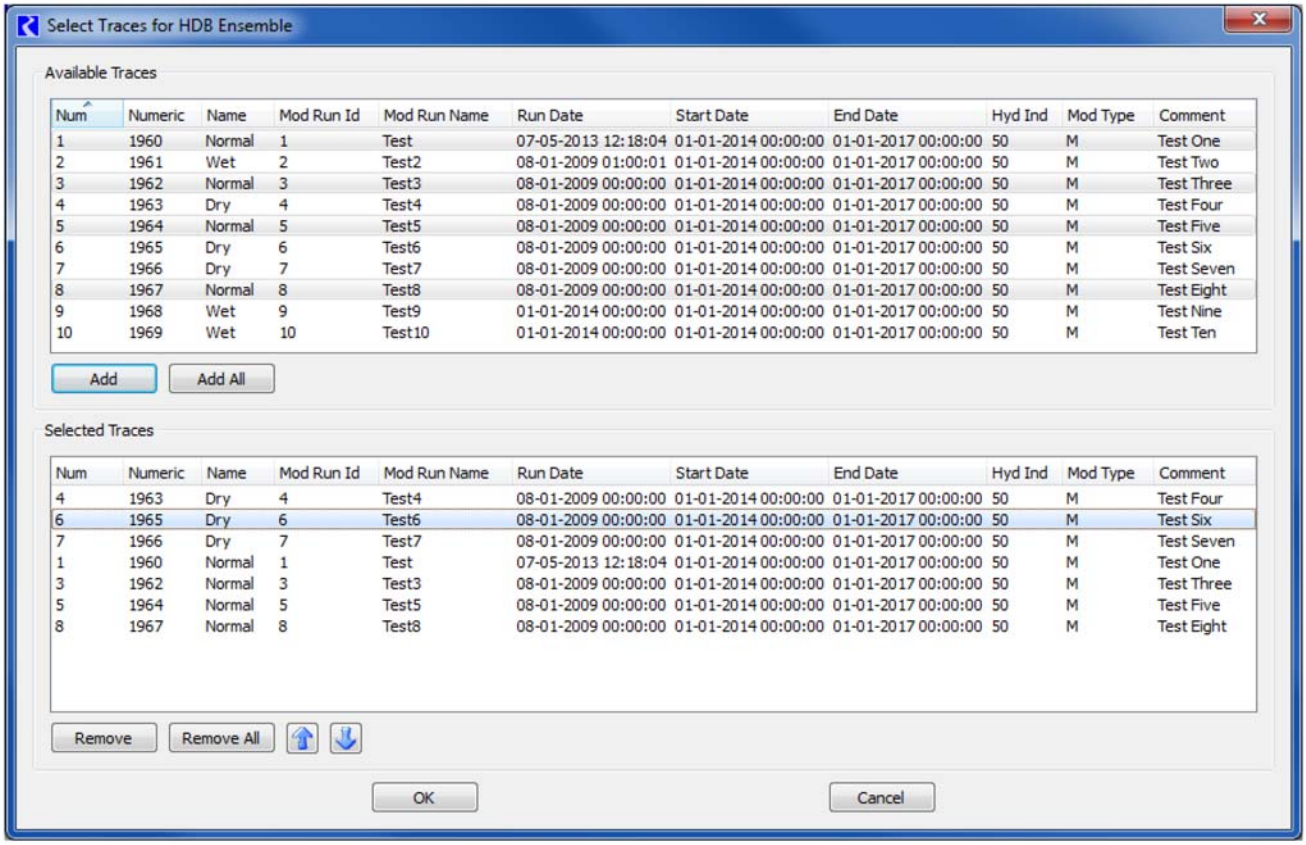
Note: Currently only HDB datasets in Database DMIs can be configured to function as ensembles. See "HDB Table Type—Ensembles" in *Data Management Interface (DMI)* for details.



Input DMIs will be executed in the order shown. This allows you to configure how data is loaded if the same slots are used in multiple input DMIs (the last one in wins). Use the up and down arrow buttons to reorder the selected DMI.

When a DMI is added as an input ensemble, it defaults to using all of the traces defined in the ensemble. The number of traces is shown for the DMI, or for its datasets in the case of database DMIs, as a column in the dialog.

When an item having traces is selected in the list, the **Select Traces** button is enabled. Select this button to open a dialog showing all of the traces for the ensemble. Choose a subset of traces as desired.



All available traces are presented in the upper list. When added using the **Add** or **Add All** buttons, they are copied to the lower list showing the selected traces. Traces can be removed from the selected list with the **Remove** or **Remove All** buttons. The selected traces can be ordered using the up and down arrow buttons. With these controls, a subset of traces in the ensemble can be chosen and used in any order as input to the multiple run. When applied with the **OK** button, the number of selected traces will now appear in the Number of Traces column in the Input Ensembles frame of the **Ensemble** tab.

The number of traces for input ensembles must match across all of the input ensembles because the number of traces input to the multiple run will determine the number of runs. If an MRM configuration with differing numbers of input ensemble traces is applied or a run is started, an error message is generated.

HDB ensemble datasets have an option to defer picking an ensemble for the dataset until the MRM run is started. See "HDB Table Type—Ensembles" in *Data Management Interface (DMI)* for details. In this case, the Number of Traces column will show a zero, and using the **Select Traces** button will return a message that the ensemble will not be selected until MRM start. Where datasets of this type are used in input ensembles, the number of runs in the multiple run cannot be determined until the MRM run is started, so the **Concurrent Runs** tab of the MRM configuration dialog will be empty. Uniformity of number of traces across input ensembles is checked after ensembles are selected at the start of the MRM run.

DMIs to use as output ensembles are added and removed with the plus and minus buttons in the Output Ensembles frame of the **Ensembles** tab. Unlike input ensembles, the order of output ensembles does not matter, so there are

no ordering controls in this frame. All existing data and metadata in output ensembles for a multiple run are cleared at the beginning of the multiple run. At the end of each run in the multiple run, the data specified in output ensemble DMIs are written to the trace of the output ensemble that corresponds to that run. An output ensemble must have at least as many traces as the number of runs in the multiple run so that all of the run data can be written. If an HDB dataset configured to select the ensemble at MRM start is used in an output ensemble, its number of traces is checked after the ensemble is selected.

Ensemble Metadata

An ensemble can have metadata that describes the ensemble as well as metadata that describes each trace in the ensemble. Metadata is represented in RiverWare as Keyword/value pairs, such as “comment”/”Climate Change Hydrology”. An important functionality of ensembles with MRM is that the ensemble and trace metadata from input ensembles are combined and copied over to output ensembles.

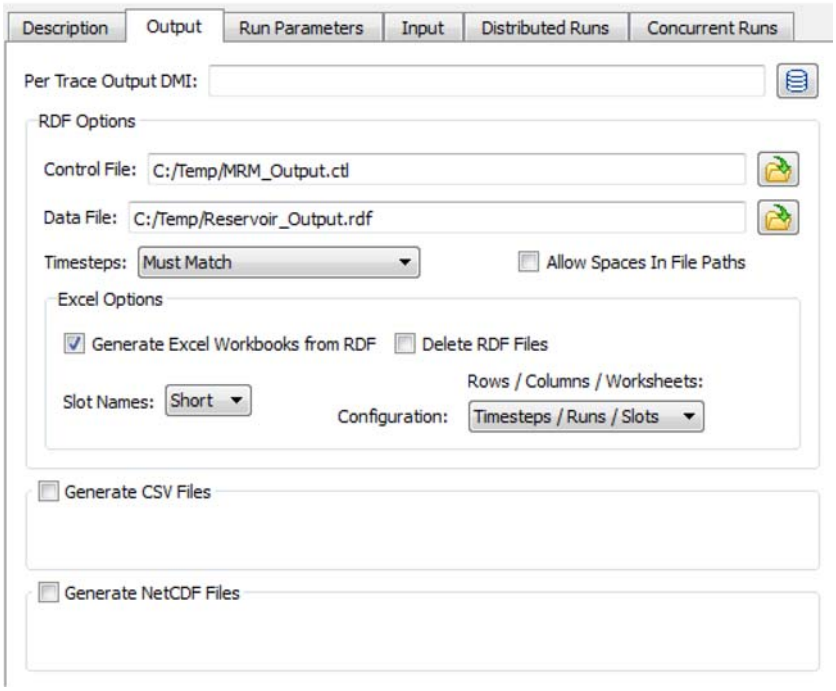
For ensemble metadata, values for the “domain”, or “comment” keywords from input ensembles are concatenated with semicolons and written as values for these keywords to output ensembles. For example, if there were two input ensembles, one with a comment “Historic Hydrology” and the other with a comment “High Projected Demand”, an output ensemble would be given the comment keyword value of “Historic Hydrology;High Projected Demand”. Values for other ensemble metadata keywords are not concatenated. If values for these other keywords differ among input ensembles, the keyword value from the first ensemble is copied to the output ensemble and a warning issued that values for this keyword differ among the input ensembles.

For trace metadata, values for the “name” keyword for multiple input ensemble traces will be concatenated with semicolons and written as the value for the “name” keyword to an output trace. Values for other trace metadata keywords are not concatenated. If values for these other keywords differ among multiple input ensemble traces, the keyword value from the trace of the first ensemble will be copied to an output ensemble trace and a warning issued that values for this keyword differ among the input ensemble traces.

See “HDB Table Type—Ensembles” in *Data Management Interface (DMI)* for details on HDB ensembles and metadata.

Output

During a multiple run, output can go to an output DMI and/or one or more RiverWare Data Format (RDF) text files. Options are also available to send output to CSV files or NetCDF files. Select the **Output** tab of the Multiple Run Editor. The following are configuration options:



DMI

The optional **Per Trace Output DMI** field allows selecting or entering an output DMI or a group of output DMIs that will be run after each single run of the multiple run. HDB ensemble datasets cannot be used in these DMIs, but rather must be used in the MRM ensemble configuration; see "Ensemble Configuration" on page 48 for details.

The DMIs must be previously configured in the DMI Manager in RiverWare. The executable that is configured for the DMI can reference the trace number of the run that is outputting by referencing the last command line parameter passed from RiverWare:

```
-STrace=traceNumber
```

Note: Using a Per Trace Excel Output DMI in combination with Distributed MRM is not recommended. In that case, multiple RiverWare processes can end up trying to write to the same Excel file simultaneously, which may cause a conflict. See "Distributed Concurrent Runs" on page 60 for details.

RDF

Configurations for RDF output from MRM include the following:

Control File

Type or use the **File Chooser** button to select a complete file path into the required control file field. The control file is used to specify which slots are output after each MRM run. The control file may contain "file =" specifiers for any line entries in the file. This causes data associated with those lines to go to the specified RDF output file. In the example control file below, if "fileName" is the same for each slot, then the output will go to a single RDF file; varying fileNames will send the output to multiple RDF files.

```
MountainStorage.Inflow: file=fileName
MountainStorage.Outflow: file=fileName
MountainStorage.Storage: file=fileName file=fileName2
```

You can optionally specify multiple files for a single slot as in the third line above. Also, slots with a timestep different than the model's timestep can be output using the same syntax.

In the control file, there are four ways to specify where the data files are created:

- Hard code the path: file=C:/DMI/Data/ResA.Inflow
- Use an environment variable: file=$(DMI_DATA)/ResA.Inflow
- Use '~': file=~/ResA.Inflow where '~' is replaced with $RIVERWARE_DMI_DIR/<DMI name>
- If the filename is not specified, the output file will be created in the directory in which RiverWare resides.

The second option is recommended as it is more transparent than '~" but still allows the model to be moved from machine to machine by setting the environment variable to the appropriate value on each machine.

Tip: See "RDF Viewer" on page 69 for more information on quickly viewing the results of an MRM run using the RDF Viewer.

Timesteps

Specify whether RDF output files may contains slots whose timesteps differ. There are three choices:

- **Must Match**. All slots written to an RDF file must have the same timestep. Slots whose timesteps differ from the file's timestep are skipped. (The file's timestep is determined by the first slot MRM associates with the file.) You are warned about slots being skipped, and asked whether to continue the MRM run.
- **Use Smallest**. Slots written to an RDF file may have different timesteps, with the output written using the smallest timestep.For example, if slots with 1 Month and 1 Year timesteps are written to an RDF file, monthly values will be written and the 1 Year slots will write 11 NaN followed by a value.
- **Use Largest.** Slots written to an RDF file may have different timesteps, with the output written using the largest timestep. For example, if slots with 1 Month and 1 Year timesteps are written to an RDF file, yearly values will be written and the 1 Month slots will write December values.

Data File

Optionally type or use the **File Chooser** button to select a complete file path into the Data File field. This file is used as the RDF output file for any lines in the control file that do not have an explicit file specifier. If the data file field is used and there are no "file=" specifiers in the control file, then all output will go to this single data file.

Excel Options

Select **Generate Excel Workbooks from RDF** to create Excel files from all of the RDF output files after all runs have completed.

Select **Delete RDF Files** to delete all RDF files after the Excel files are created. (The separate RdfToExcelExecutable program contains the same functionality for creating Excel files from RDF files and can be used outside of RiverWare to process RDF files from an MRM run.)

Select a **Configuration** from the menu to control how timestep, slot, and run data from RiverWare are mapped onto the Excel dimensions of rows, columns, and worksheets.

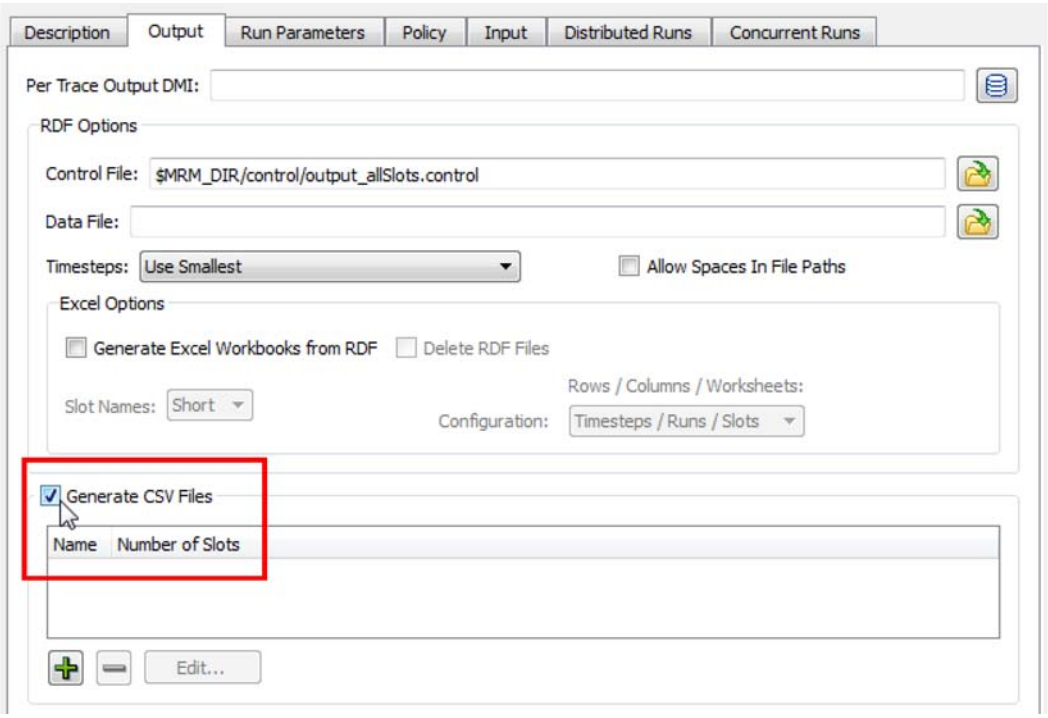
Select an option from the **Slot Names** menu to specify how slot names are written into the Excel workbook. Options are as

- **Index**: (Slot0, Slot1, etc.)
- **Short**: Automatically shortened names (lower case vowels removed)
- **Full**: Full slot names (limited to 31 characters for worksheet names, which is the limit for Excel)

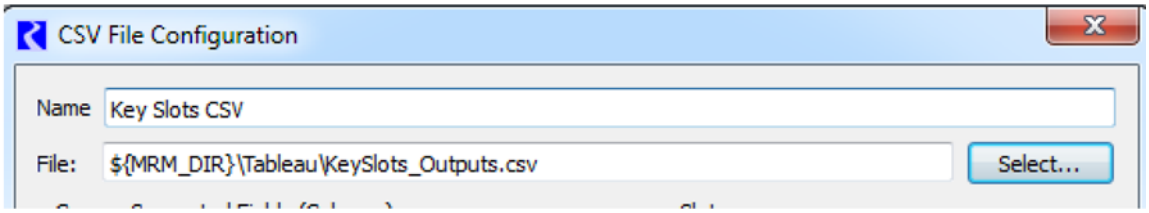
CSV Files

CSV files can be generated from the MRM run outputs. The CSV files are formatted for direct use within Tableau data visualization software. The configuration allows for the selection of various pieces of information to be used as dimensions in Tableau. The CSV output is essentially a table with the columns delimited by commas.

1. Near the bottom of the **Output** tab in the MRM configuration dialog, select the checkbox for **Generate CSV Files**.

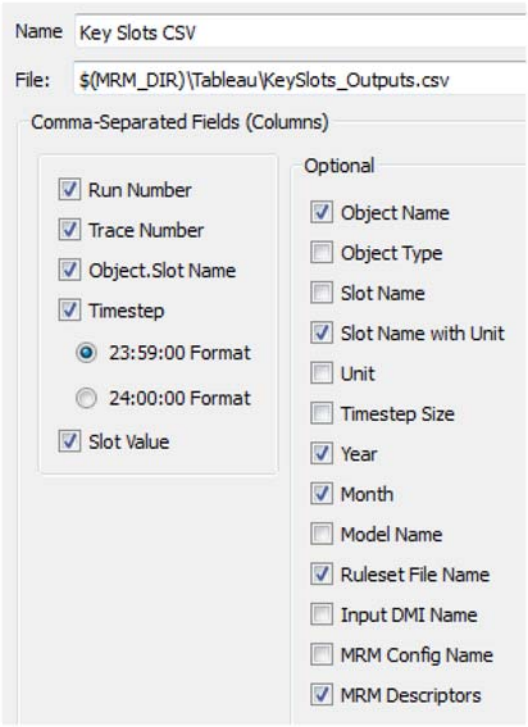


2. Select the **Plus** sign to add a new CSV output file. Then select **Edit** to configure its contents. This will open the CSV File Configuration dialog.
3. The CSV can then be given a name to display in the list in the MRM configuration.
4. In the **File** field, either select a CSV file to which the new outputs should be written by selecting the **Select** button, or type in a complete file path. Environment variables can be used in the file path preceded by a "$".



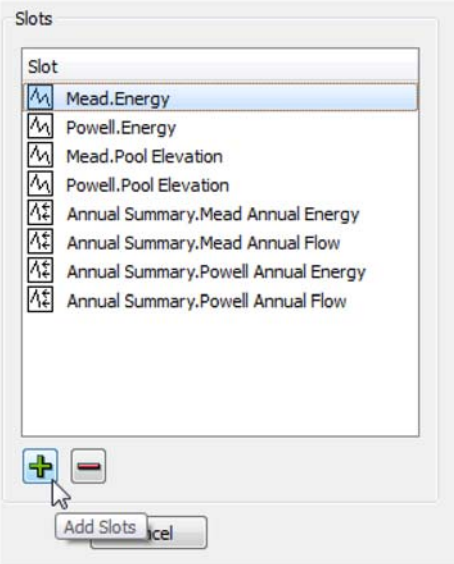
5. Next, select which pieces of additional information should be included as columns in the output table by selecting or clearing the checkbox by each optional element. Figure 3.5 shows the list of available elements. The CSV output file will contain a column for each item selected. The text shown in the CSV File Configuration dialog will be used as the column header. The Slot Value column is the only column that will contain true "output" data from the MRM run. The Slot Value will be considered a "measure" within Tableau. The remaining columns contain information associated with each Slot Value and will be considered *dimensions* within Tableau.

Figure 3.5



Note: In the context of Distributed Multiple Runs the Run Number corresponds to the run number on the individual processor. Thus if there are eight runs distributed to four processors (two each), all rows in the resulting CSV file will show a Run Number of either 1 or 2.

6. Then add slots to the CSV output by selecting **Plus** sign below the Slots panel. This will open a slot selector dialog, which can be used to add the desired slots. A slot can be removed from the selection by selecting it in the list to highlight it, then selecting the **Minus** sign.



7. After configuring the CSV output, select **OK** in the CSV File Configuration dialog, and select **Apply** or **OK** in the MRM Configuration dialog to apply the changes.

During the MRM run, RiverWare will write the outputs to the specified CSV file. The file will contain one row for each selected slot at each timestep for each run. For each row, the columns will be filled in appropriately by RiverWare based in the selected dimensions. Figure 3.6 shows part of a sample CSV output file (viewed in Microsoft Excel).

Figure 3.6

	A	B	C	D	E	F	G	H
1	Trace	Object.Slot	Timestep	Slot Value	Object	Slot Name with Units	Year	Month
1600	5	Powell.Pool Elevation	11/30/2022 23:59	3578.06	Powell	Pool Elevation (ft)	2022	November
1601	5	Powell.Pool Elevation	12/31/2022 23:59	3571.91	Powell	Pool Elevation (ft)	2022	December
1602	1	Mead.Energy	1/31/2013 23:59	325.579	Mead	Energy (GWH)	2013	January
1603	1	Mead.Energy	2/28/2013 23:59	312.387	Mead	Energy (GWH)	2013	February
1604	1	Mead.Energy	3/31/2013 23:59	346.615	Mead	Energy (GWH)	2013	March
1605	1	Mead.Energy	4/30/2013 23:59	363.225	Mead	Energy (GWH)	2013	April

RiverWare uses end of timestep format for all datetime references. Thus for a 1 Month timestep, July 2014 would be fully specified as 7/31/2014 24:00. Tableau and Excel do not have a concept of 24:00 as a time but would instead use 8/1/2014 00:00 for the same point in time. Therefore with the proper timestep option selected in the dialog, one minute is subtracted from all timestep values (e.g. 7/31/2014 23:59) to assure that they appear with the appropriate day and month references in Excel and Tableau.

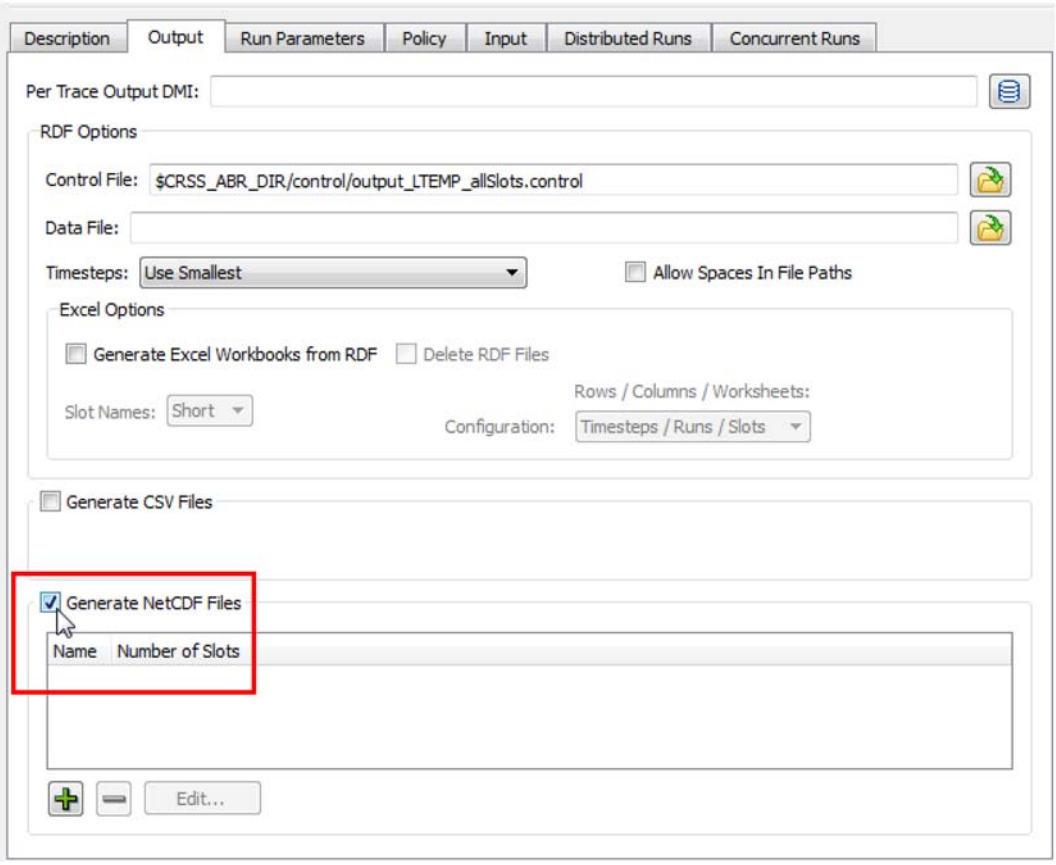
> **Note:** If the specified CSV file already exists, the CSV output from the new MRM run will overwrite the existing CSV file. It will not append data to the existing file.

NetCDF Files

Network Common Data Format (netCDF) files can be generated from the MRM run outputs. The configuration allows for the selection of various pieces of information to be used as global attributes and slot (variable) attributes. Time and Traces are treated as dimensions.

> **Note:** RiverWare will write the file in the netCDF-3 format, with one unlimited dimension (time). The output does not use any of the features available in netCDF-4 and testing has shown that netCDF-3 provides a much smaller file size than using netCDF-4.

1. Near the bottom of the **Output** tab in the MRM configuration dialog, select the checkbox for **Generate NetCDF Files**.



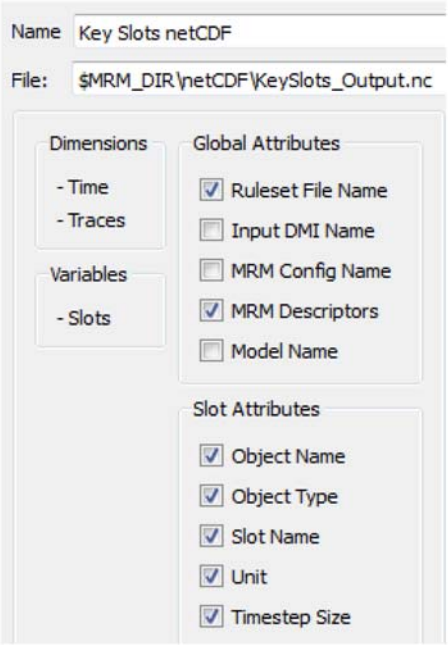
2. Select the **Plus** sign to add a new netCDF output file. Then select **Edit** to configure its contents. This will open the NetCDF File Configuration dialog.
3. The netCDF file can then be given a name to display in the list in the MRM configuration.

4. In the **File** field, either select a netCDF file to which the new outputs should be written by selecting the **Select** button, or type in a complete file path. Environment variables can be used in the file path preceded by a "$".



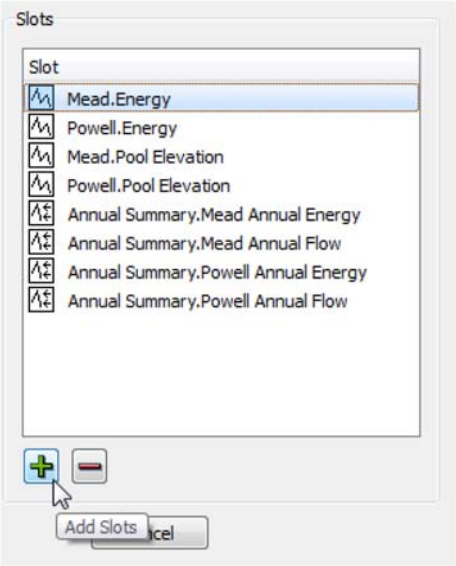
5. Next, optionally select global attributes and slot attributes to be included with the outputs by selecting or clearing the checkbox by each element.

   Note: Time (timestep) and Traces (Trace Number) will automatically be used as dimensions in the netCDF output.



6. Then add slots to the netCDF output by selecting the **Plus** sign below the Slots panel. This will open a slot selector dialog, which can be used to add the desired slots. A slot can be removed from the selection by

selecting it in the list to highlight it, then selecting the **Minus** sign. The Object.Slot name will be used as the variable name in the netCDF file.



7. After configuring the netCDF output, select **OK** in the NetCDF File Configuration dialog, and select **Apply** or **OK** in the MRM Configuration dialog to apply the changes.

Saving and Restoring Initial Model State

This section describes how to save and restore the initial state of a model. The functionality allows users to save model data to a temporary file before a multiple run is invoked and then restore this information after the run has completed.

- Open the MRM Dialog by selecting **Control**, then **MRM Control Panel** from the main RiverWare menu bar or select the **MRM** button on the toolbar. 
- Save the initial state of a model before invoking a MRM run by pressing the **Save Initial State** button in the Model State area of the MRM Control Panel.
- Start the MRM run by pressing the **Start** button.
- After the run has finished, restore the initial model state by selecting the **Restore Initial State** button.

Selecting the **Save Initial State** button saves the entire contents of the model, including both input and output slots, TableSlots, selected user methods, run information, and configuration to a file in your temporary directory (defined by the TMP environment variable or system temporary variable as shown in the **Help**, then **About** RiverWare menu, then select the **Show System Info** button). Once the initial state has been saved, the button is disabled and reads “Initial State Saved.”

- **Save Initial State** button prior to saving state.

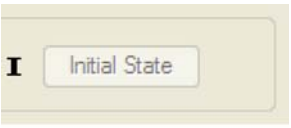


- **Save Initial State** button after saving state.



Once an initial state has been saved, it can be restored by selecting the **Restore Initial State** button. This reloads the entire contents of the model at the time its state was saved, clearing all data and configuration in the current state of the model. Until a multiple run has occurred, this button is disabled and reads “Initial State”.

- **Restore Initial State** button before a multiple run.



- **Restore Initial State** button after a multiple run.



Distributed Concurrent Runs

When performing concurrent MRM runs consisting of many runs, the following are some problems that users encounter:

- The memory required may exceed the resources available and/or
- The time required to make the runs is excessive.

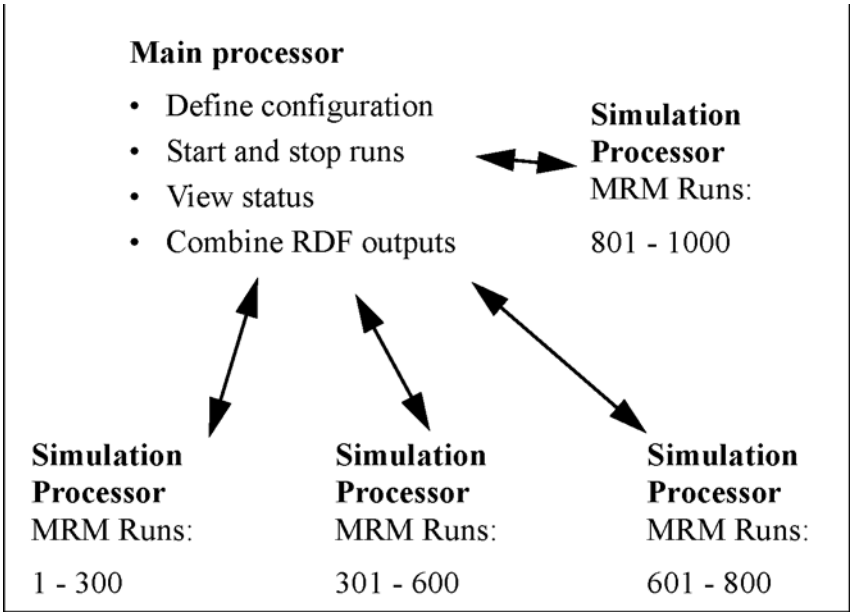
This section describes a utility that solves these two issues by distributing many runs across many processors on the same machine.

> **Note:** All runs are made on the same computer that has multiple cores or multiple logical processors. In this document, each separate MRM instance is referred to as a *simulation processor*.

In this approach, there is a controller processor and simulation processors. The controller processor does the following:

1. Creates the configurations
2. Controls execution (Start and Stop) of each simulation.
3. Tracks the progress.
4. When all the runs are finished, combines the output RDF file from each simulation into one output RDF file.

Figure 3.7



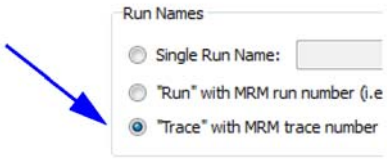
Each simulation processor then executes the MRM runs that the controller gives it to execute. Each processor runs one or more MRM runs that consist of a portion of the total number of concurrent runs. This is shown graphically in the screenshot. In this example, there are 1000 runs that are distributed unequally to four processors.

> **Note:** If you are running RiverWare using a floating license, you can only run as many RiverWare sessions as your license allows. For example, a three-seat floating license can only run 3 RiverWare sessions. Therefore, you can distribute an MRM run to only three processors. If you wish to distribute to more processors, check out a roaming license. Node-locked and roaming licenses can run unlimited RiverWare sessions. The license guides can be found at the following URL:
>
> `http://www.riverware.org/users/`

This feature was initially designed for a very specific application of concurrent MRM. As such, the following are the limitations and constraints:

- The same version of RiverWare must be used for each run.
- In the MRM configuration, the Inputs must be Traces or Index Sequential with the Pairs mode selected. See "Combined Runs" on page 47 for details.
- Input DMIs can include Control File-Executable or Excel Database DMIs with the Header approach. If using Excel, you must use the **Trace** configuration option for the Run Names. The **Run** option is not supported.

Figure 3.8



- Only a single ruleset may be used.

This document is organized to present an overview of the user interface, how to make a run, and how the utility works to distribute the runs.

User Interface Overview

The interface for distributing MRM runs across multiple simulations consists of two components, the MRM configuration within RiverWare and the Distributed MRM dialog which is external to RiverWare.

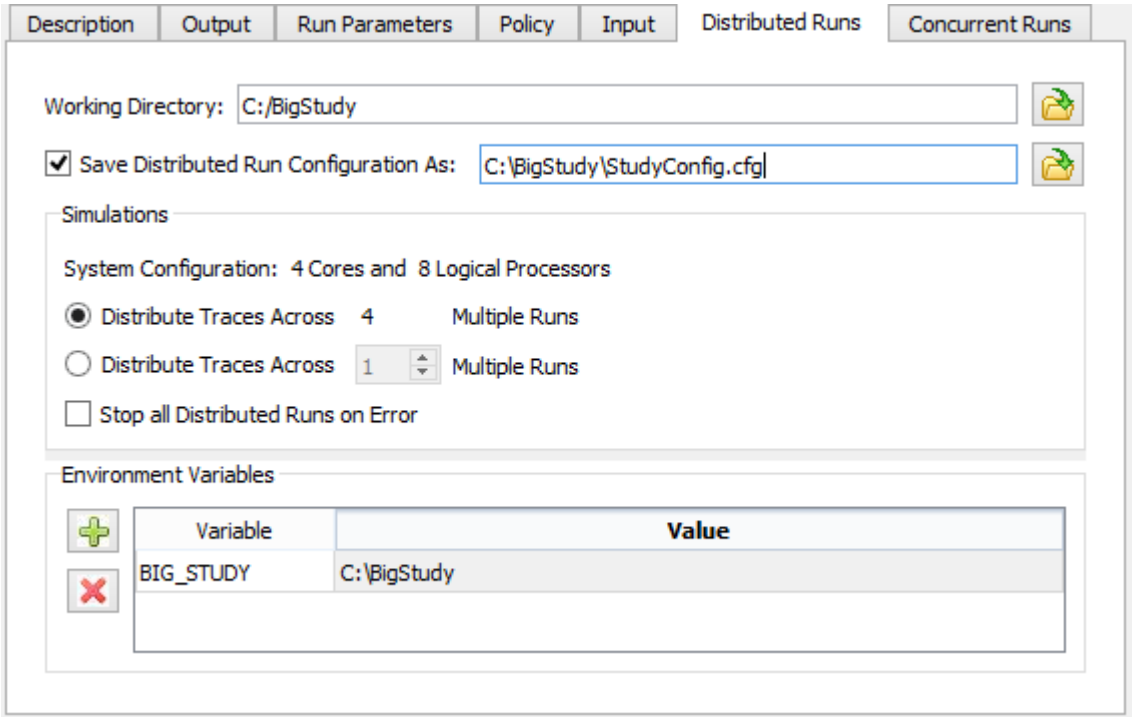
MRM Configuration

MRM runs can only be distributed when in Pairs mode; see "Combined Runs" on page 47. Thus, when in Pairs mode, the **Input** tab has a **Distributed Runs** checkbox that becomes enabled. When checked, the **Distributed Runs** tab is added to the dialog.



As discussed in the sections that follow, the **Distributed Runs** tab allow you to configure:

- The working directory.
- Whether the configuration should be saved to a file.
- The number of multiple runs to distribute the traces across.
- Environment variables and their values.



Working Directory

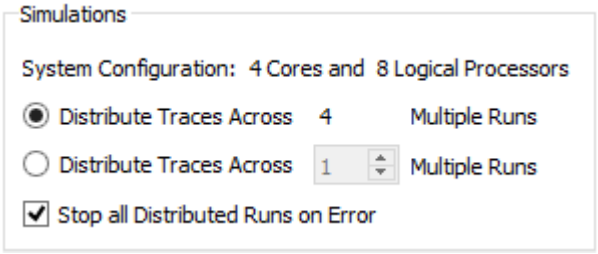
A distributed concurrent run creates several *working* files - batch script files, control files, intermediate RDF files and log files among them. The working directory specifies where the working files will be created. Importantly, it should be a directory which the controller processor and all simulation processors have access to (via the same path).

Save Configuration As

When a distributed concurrent run is started, RiverWare writes a configuration file, invokes the Remote Manager process, and exits. If a user elects to save the configuration to a named file, it is possible to invoke the Remote Manager directly bypassing RiverWare. See "Creating or Changing a Configuration" on page 65 for details.

Simulations

The Simulations section defines the simulation processors and the traces they will simulate.



The System Configuration line provides the number of cores and the number of logical processors on the machine. You can distribute your runs as follows:

- **Distribute Traces across N Multiple Runs**. Use all available cores on the machine. The model can be moved to different computers and the configuration will use all of the cores.
- **Distribute Traces across <N> Multiple Runs.** Specify the distribution. Specify a set number of runs so the runs complete in a reasonable amount of time without using all of a computer's resources. Or if you have a lot of RAM, you may specify a larger number, likely up to the number of logical processors.

The choice is ultimately hardware and model dependent. To make the best choice you will need to have detailed knowledge of your model's memory usage and hardware. Tools like SysInternals may help to identify the maximum working set when running the model. Experimentation may be necessary to determine the most efficient settings.

Finally, in this section is a control to **Stop all Distributed Runs on Error**. When checked, if one distributed run aborts, the remaining distributed runs will also be stopped. This prevents having to wait for all runs to finish when one has failed and the results are not valid.

Environment Variables

You can enter environment variables and value pairs.

Remote Manager and Status Dialog

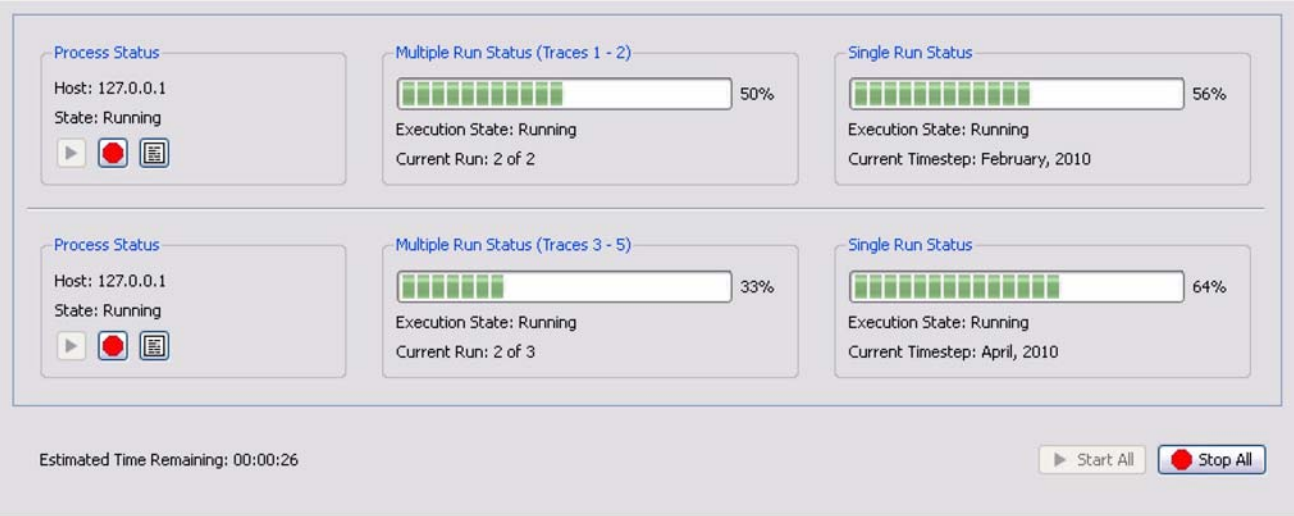
As mentioned above, the Remote Manager includes a user interface which displays the status of the simulations. This dialog is not within RiverWare but is a separate executable in the installation directory. It is also opened

automatically when you make a distributed MRM run from the RiverWare MRM Run Control. More specifically, the Remote Manager user interface allows you to:

- Start and stop the simulations individually or collectively.
- View RiverWare's diagnostic output for the simulations.
- View the multiple and single run status.
- See an estimated time remaining.
- See the status of the post-processing (combining the RDF files).

In Figure 3.9, from left to right, are the Process Status, Multiple Run Status, and Single Run Status panels.

Figure 3.9



The Process Status panel includes the **Start**, **Stop**, and **View Diagnostics** buttons. The buttons allow you to start, stop, and view diagnostics for the individual run/process.



The Multiple Run Status and Single Run Status panels (which are very similar to RiverWare's run status dialog) display the progress of each MRM run and each individual run:



The bottom of the dialog shows the estimated time remaining based on available data after the first run has completed.



How to Make a Distributed Run

There are two options, creating/changing the configuration (within RiverWare) or rerunning a configuration (can be external to RiverWare). These are described in the next two sections:

Creating or Changing a Configuration

If you are creating a new configuration or changing an existing configuration, the changes must be made from within RiverWare. This will allow RiverWare to create the necessary configuration files that will be passed to the simulation controllers. When you select **Start**, RiverWare will create the necessary configuration files, start the RiverWare Remote Manager, and then exit. The RiverWare Remote Manager controls the execution of the MRM runs.

Use the following steps to make the runs in this case:

1. Open RiverWare.
2. Fully define the configuration or make any changes to an existing configuration in the MRM Run Control Configuration dialog. See "User Interface Overview" on page 62 for the options.
3. Apply the changes.
4. SAVE THE MODEL. The distributed runs open and run the model that is saved on the file system. Therefore, you should save the model now, so that the configuration is preserved. Any changes to the model (including external files such as the RDF control file) will invalidate the saved configuration file.
5. Select **Start** on the Multiple Run Control Dialog. RiverWare will start the Remote Manager and then start the shutdown sequence. It will prompt you for confirmation so you can cancel at any time.
6. From the Remote Manager, select **Start** to start the distributed runs. See "Remote Manager and Status Dialog" on page 63 for details.
7. The individual runs start and the status is shown including an estimate of the time remaining.

8. When all runs are complete, the output RDF files are combined into one final RDF file.

Rerunning a Configuration

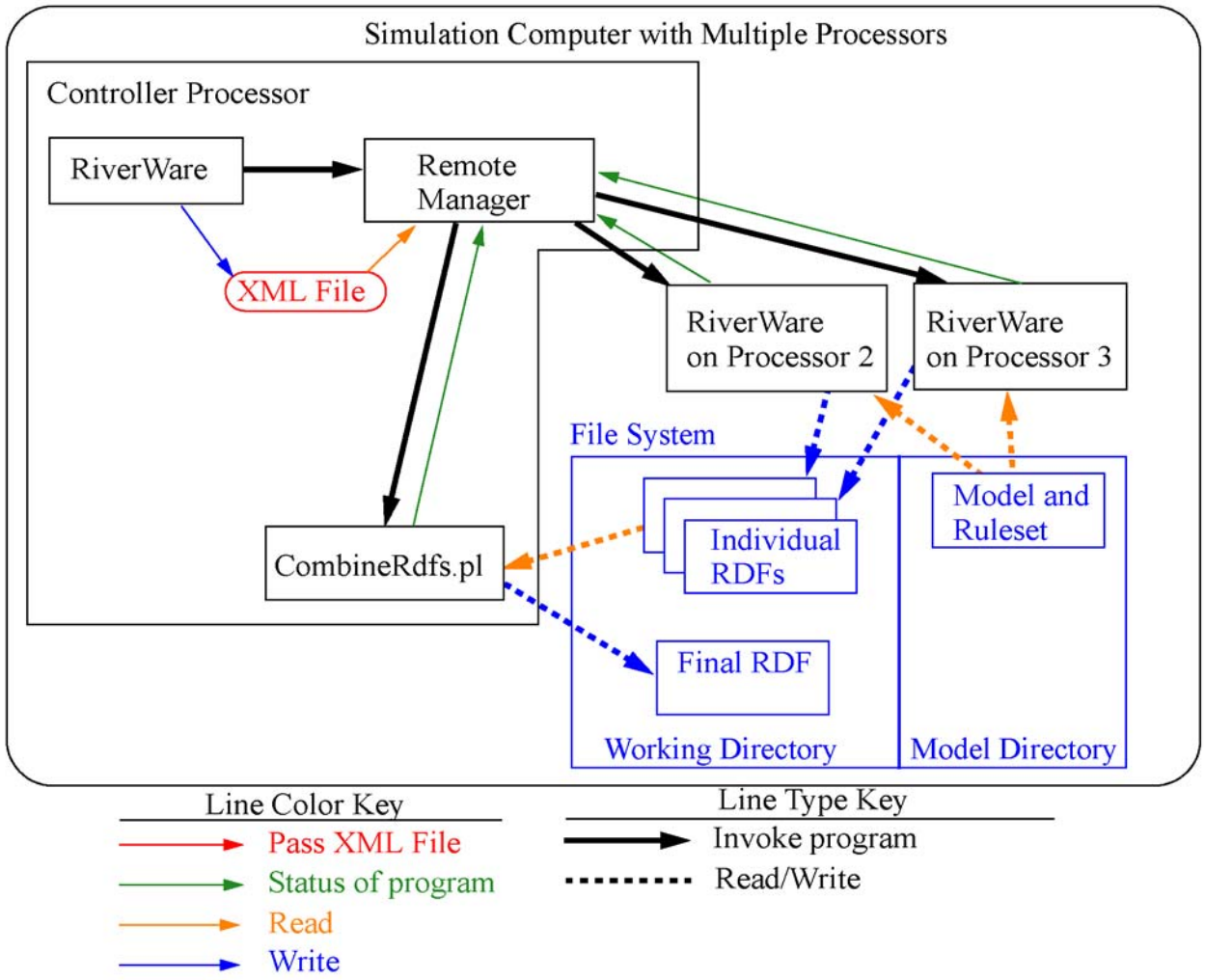
If you are repeating a previously saved configuration, then you can execute the RiverWare Remote Manager directly and not open the RiverWare. In that case, you start at Step 6.. The Remote Manager will first open a simple dialog allowing you to select the configuration file and will then open its user interface. See "Save Configuration As" on page 63 for details.

How It Works

Figure 3.10 illustrates the distributed architecture. In the distributed architecture there is a *controller* processor and one or more *simulation* processors.

The distributed architecture includes two processes - the RiverWare Remote Manager (RwRemoteMgr.exe) and instances of RiverWare (RiverWare.exe).

Figure 3.10



RiverWare Remote Manager

The RiverWare Remote Manager parses an XML configuration file that defines the simulations and:

- Creates an XML configuration for each of the simulations.
- Configures its user interface (a dialog which shows the status of the simulations).
- For each simulation, connects to the simulation processor, writes the XML configuration, and reads RiverWare's output (which it uses to update its status dialog).
- When all simulations have finished, combines the partial RDF files to create the final RDF files.

XML Configurations

Previous sections have referred to XML configurations. These are the files that RiverWare creates to control/define the distributed MRM runs.

> **Note:** This section is a technical reference providing more information on how this utility works. The XML files are generated by the utility and the user does not need to edit these files to make a distributed MRM run.

The top-level XML configuration <distrib> identifies a distributed concurrent run and is hereafter referred to as the *distributed configuration*. An XML document defining a distributed concurrent run would have the following top-level elements:

```
<document>
  <distrib>
  ...
  </distrbv>
</document>
```

The XML document defining a distributed concurrent run is created by RiverWare when a user starts the run. DistribMrmCtlr parses the distributed configuration and creates a configuration for each of the simulations. Some elements are from the MRM configuration, others are provided by RiverWare. Some elements are common to all simulations, others are unique to each simulation.

Example 3.1 Sample XML File

Key elements in this example are preceded by brief descriptions.

```
<distrib>
```

The RiverWare executable which started the distributed concurrent run. The assumption is that all simulations use the same executable.

```
  <app>C:\Program Files\CADSWES\RiverWare 6.4\riverware.exe</app>
```

The model loaded when the user starts the distributed concurrent run.

```
  <model>R:\\CRSS\\model\\CRSS.mdl</model>
```

The global function sets loaded when the user starts the distributed concurrent run.

```
  <gfslist>
    <gfs>R:\\CRSS\\model\\CRSS.gfs</gfs>
  </gfslist>
```

The config attribute is the MRM configuration selected when the user starts the distributed concurrent run. The mrm elements are from the MRM configuration and identify for each simulation the traces to simulate.

```
  <mrmlist config="Powell Mead 2007 ROD Operations">
    <mrm firstTrace="1" numTrace="200"/>
    <mrm firstTrace="201" numTrace="200"/>
  </mrmlist>
```

The rdflist element is a list of the final RDF files, while the slotlist element is a list of the slots which are written to the RDF files. Slots can be written to multiple RDF files; they're associated with the RDF files by the idxlist attribute, whose value is a comma-separated list of RDF file indices. RiverWare initializes the RDF DMI and mines its data structures to generate rdflist and slotlist.

```
  <rdflist num="2">
    <rdf name="R:\\CRSS\\results\\Res.rdf" idx="0"/>
```

```
    <rdf name="R:\\CRSS\\results\\Salt.rdf" idx="1"/>
  </rdflist>
  <slotlist>
    <slot name="Powell.Outflow" idxlist="0,1"/>
    <slot name="Powell.Storage" idxlist="0"/>
  </slotlist>
```

The envlist element specifies RiverWare's runtime environment; RIVERWARE_HOME is from the version of RiverWare which starts the distributed concurrent run, all others are from the MRM configuration.

```
  <envlist>
    <env>RIVERWARE_HOME_516=C:\\Program Files\\CADSWES\\RiverWare 5.1.6 Patch</env>
    <env>CRSS_DIR=R:\\CRSS</env>
  </envlist>
```

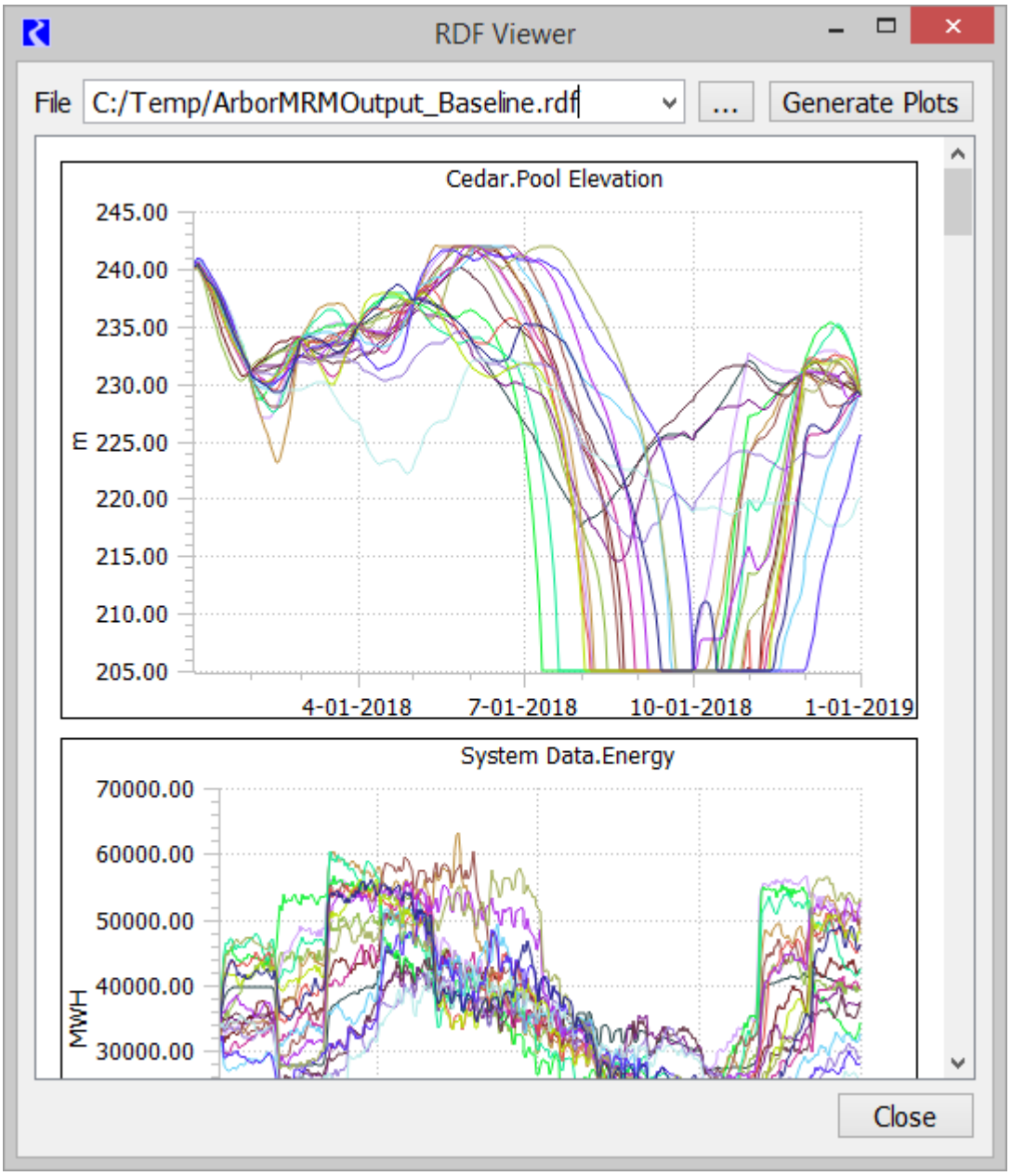
The tempdir element is from the MRM configuration and is the intermediate directory where the individual simulations write the partial RDF files.

```
  <tempdir>R:\\CRSS\\temp</tempdir>
</distrib>
```

RDF Viewer

The RDF Viewer is a simple tool for quickly visualizing the RDF results of an MRM run. A sample RDF file is shown as over plotted lines in Figure 3.11. This section describes the RDF Viewer and how to use it.

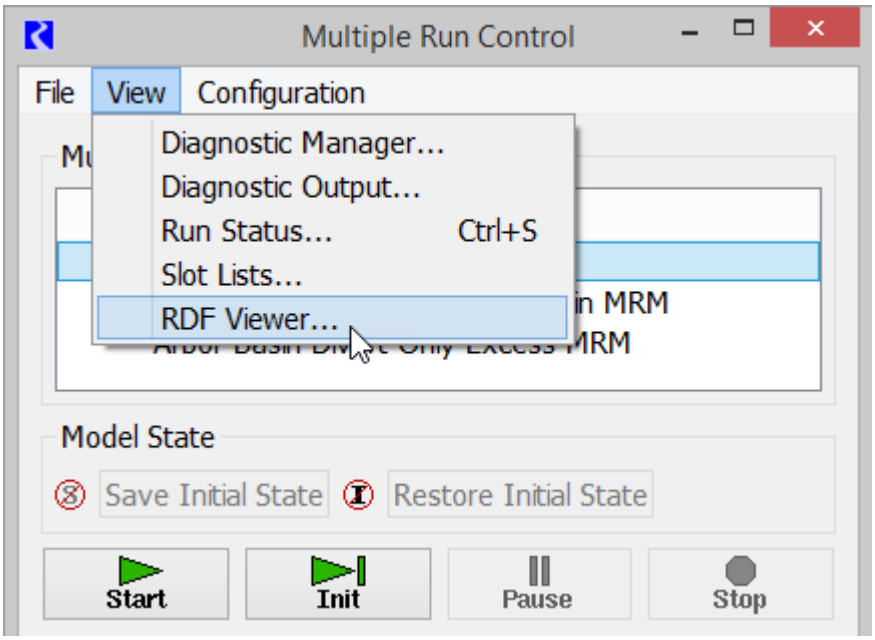
Figure 3.11 Screenshot of the RDF Viewer



Accessing the RDF Viewer

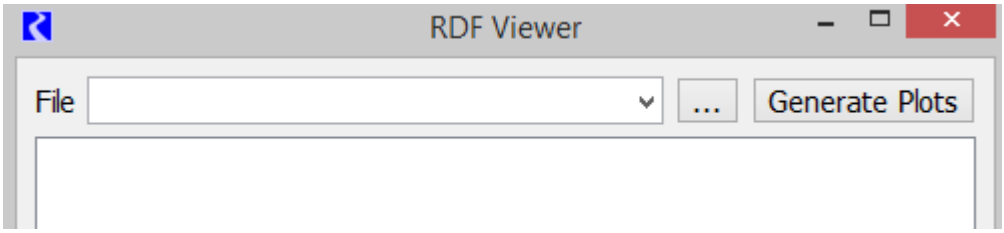
Access the RDF Viewer from the Multiple Run Control dialog using the View and then RDF Viewer menu as shown in Figure 3.12.

Figure 3.12 Multiple Run Control with RDF Viewer menu



Then the RDF Viewer opens as shown in Figure 3.13.

Figure 3.13 RDF Viewer with no file specified

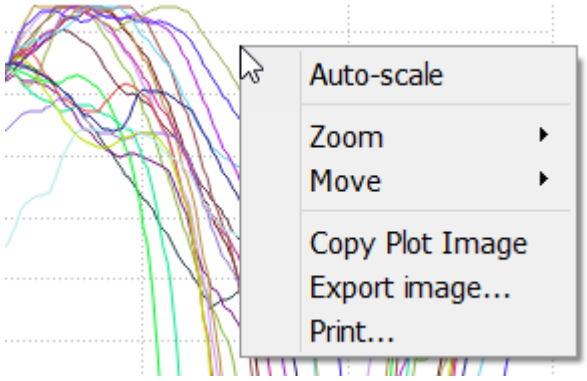


How to Use the RDF Viewer

Within the RDF Viewer, type a file path and name or use the ellipsis button to open a file chooser, then select the desired RDF file and click Open. The RDF Viewer will read the RDF file and automatically generate plots of the slots. The tool displays plots of the RDF data, one plot per slot, with runs (traces) over plotted as shown in Figure 3.11

Hover over a curve to see the run/trace number. Right click on a plot area for additional plotting functionality as shown in Figure 3.14. In addition, use the mouse wheel to zoom in/out on either the plot area or the axis. Click and drag the middle mouse button to pan the plot.

Figure 3.14 RDF Viewer Plotting controls.



Chapter 4
Accounting

River and reservoir basins are operated for various purposes, including power production, irrigation, environmental purposes, recreation, and water quality. In many basins, it is necessary for water managers to track not only the volume and flow of water throughout the basin, but managers must also track the water ownership and type of the water. Operating decisions in the basin are dependent on many factors, including a user's available water, legal restrictions, physical constraints, and any exchange mechanism. In addition, many of the basins in the United States are governed by the doctrine of Prior Appropriation which says that the right to use a water supply is based on "first in time, first in right". Because of these constraints, it is necessary for water managers to have a tool to simulate the operating decisions and their effect in the basin including water type and water ownership.

RiverWare meets these requirements through water accounting. Simply, water accounting is a layer added to a simulation model used to track the ownership and/or type of the simulated water. In an accounting model, accounts, slots, and data are added to track why water is released, stored, or diverted. This network is separate from the simulation network but there are methods that allocate physical water to the accounting network.

Accounts are linked to one another on both the same object and on different objects. These links indicate the possible transfers of water in the system. By defining a link, the user indicates that at some timestep, water could move between the two linked accounts. Legal accounts, including Storage, Diversion, and Instream Flow, are used to model a legal water right. Passthrough accounts are used to track the transfer of water through basin features. With both legal and non-legal accounts, the user can look at any object in the basin and determine the type and ownership of all of the water in that object.

Accounts solve whenever sufficient known slot values are present to run a solution method. This is analogous to dispatching in the simulation network; however, in the case of the accounting system, this means that accounts may solve when a user edits the account values through the user interface, a rule sets a value, or a method sets a value. Therefore, a run is not required to make a single account solve. Each time an account slot is given a value, the account holding the slot is notified, and performs its own checks for over- and under-determination, and possibly solves at one or more timesteps.

Outside of a run, accounts solve for as many timesteps as there is the required known data. This means that the account solution behaves similarly to a spreadsheet solution. If the user inputs an Outflow for each timestep in the accounting period for a storage object and all of the inflows, slot inflow and Gain Loss are known, then the account will solve for Storage at each timestep.

Rules can access the account solution but can also control and set the releases from the accounts. User inputs can also drive the account solution. Using rules and other accounting utilities, the user can simulate typical accounting functionality including accrual, exchanges, carryover, and allocation. In addition, a special predefined rule function and set of methods serves as a water rights allocation solver to allocate water to legal accounts based on prioritized water rights.

For details about the RiverWare Accounting, see "Accounting Overview" in *Accounting*.

Chapter 4
Accounting

Chapter 5
Optimization

The RiverWare Optimization solver uses a preemptive linear goal programming solution. Instead of solving a model on a timestep-by-timestep basis, Optimization provides a single global solution over all objects and all timesteps. Typically Optimization is used to determine the "best" solution in terms of a stated objective, such as maximizing the value of hydropower over the run period, while satisfying higher-priority policy constraints.

For details about the RiverWare Optimization solver, see "Optimization Overview" in *Optimization*.

Chapter 5
Optimization