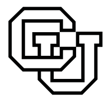
Technical Documentation Version 6.2

Simulation



CADSWES

Center for Advanced Decision Support for Water and Environmental Systems

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.

Simulation
Table of Contents

Introduction 1

Mathematical Basis for Reservoir Simulation 1

Dispatch Methods 2

User Method Dependent Dispatch Methods 3

Slot Values 3

Set Value 4
Object's Response to Setting a Slot Value 4
Iteration 5

Simulation Controller 5

Initialization 6
First Dispatch Timestep 6
Initialization Rules.. 6
Time Horizon Simulation 8
Under-determination 9

Object Dispatching Patterns 9

2-Reservoirs with Tailwater-dependent Energy 9
Bi-Directional Canal 10
Reach and AggDiversionSite Interaction 10

Simulation - How it Works

1. Introduction

The Simulation Controller has the task of managing 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.” This section discusses:

- The mathematical basis for Reservoir Simulation
- Dispatch Methods
- Propagation of Values
- Iteration and Convergence
- Dispatching Patterns

2. 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:

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

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

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

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

$$TotalInflow = Inflow + HydrologicInflow$$

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.

3. 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. Listed below are the some of the Dispatch Methods and their conditions for the Level Power Reservoir object:

solveMB_givenInflowOutflow

Knowns:	*Unknowns*:
Inflow	Storage
Outflow	Pool Elevation

solveMB_givenInflowStorage

Knowns:	*Unknowns*:
Inflow	Pool Elevation
Storage	Outflow
	Energy

solveMB_givenOutflowStorage

Knowns:	*Unknowns*:
Outflow	Pool Elevation
Storage	Inflow

solveMB_givenInflowHW

Knowns:	*Unknowns*:
Inflow	Storage
Pool Elevation	Outflow
	Energy

solveMB_givenOutflowHW

Knowns:	*Unknowns*:
Outflow	Storage
Pool Elevation	Inflow

solveMB_givenOutflowStorage

Knowns:	*Unknowns*:
Outflow	Pool Elevation
Storage	Inflow

solveMB_givenEnergyStorage

Knowns:	*Unknowns*:
Energy	Pool Elevation
Storage	Inflow

solveMB_givenEnergyHW

Knowns:	*Unknowns*:
Energy	Storage
Pool Elevation	Inflow

solveMB_givenEnergyInflow

Knowns:	*Unknowns*:
Energy	Pool Elevation
Inflow	Storage

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.

3.1 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 following Dispatch Method is only added to the dispatch table when the **solveHydrologicInflow** User Method is selected:

solveMB_givenInflowOutflowStorage

Knowns:	*Unknowns:*
Inflow	Pool Elevation
Outflow	Hydrologic Inflow
Storage	

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

4. 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 three mechanisms:

1. Direct user input.
2. A value propagated across a link from another object.

3. A value set by the object’s user or dispatch methods.

4.1 Set Value

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

- 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. For more information on how to set convergence for a series slot, click HERE (Slots.pdf, Section 3.1.1).

- 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 ➥ Simulation Run Parameters** from the workspace (HERE (RunControl.pdf, Section 2.1)).

- 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.

- Set Value

If the previous checks succeed, set the (new) value in the slot.

- Propagation

If the slot is linked, propagate the value across the link(s).

4.2 Object’s Response to Setting a Slot Value

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

- 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 re-dispatch when they are set. If the slot is not a dispatch slot, nothing else is done.

- 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, re-dispatch with the same Method.

- 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.

4.3 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 re-dispatches using the same Dispatch Method that was invoked previously.

5. Simulation Controller

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

- Initialization for simulation
  - Clear all output and values from previous runs for all timesteps in all series slots.
  - Set user inputs.
  - Propagate user inputs across link.
  - Determine first dispatch timestep. Click HERE (Section 5.1) for more information.
- Execute rules in the Initialization Rules RPL set. Click HERE (Section 5.1.2) for more information.
- Simulation Beginning of Run
  - Execute Beginning of Run methods for all objects.
  - Evaluate Beginning of run expression slots for all timesteps.
- For each timestep:
  - Set the controller clock to the timestep time.
  - Execute Beginning of Timestep methods for all objects.
  - Evaluate Beginning of timestep, current timestep only expression slots
  - Dispatch objects until the queue is empty, simulating the effects of the user inputs and default values.
  - Execute End of Timestep methods on all objects.
  - Evaluate end of timestep, current timestep only Expression slots
- Execute End of Run Simulation methods on all objects.
- Evaluate End of Run expression slots.

5.1 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.

5.1.1 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 pre-run timesteps. This enables pre-run inputs to route downstream thus allowing the system to solve on the start timestep.

Finding Inputs: For Reaches, Control Points and Confluences, 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 or Confluence 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 pre-run timesteps necessary for downstream lagging. 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. Click HERE (Objects.pdf, Section 7.1.24) for more information on these methods.

5.1.2 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 the other mechanisms for providing input to a run, that is, to interactive setting of user input and execution of DMIs.

This section describes the applicability and use of Initialization Rules. The user interface for Initialization Rules is described in general with the other RPL sets HERE (RPLUserInterface.pdf, Section 1).

Applicability (when to use Initialization Rules): Each method for providing data to a simulation is suited to different modeling scenarios. For situations in which there are a small number of inputs that don't change frequently, interactive setting of input values is often the simplest approach. However, when this process would be time-consuming or error-prone, it is preferable to automate the setting of user inputs. Input DMI's provide a flexible way of accomplishing this by providing mechanisms for communication between RiverWare and external programs (usually databases). If there is

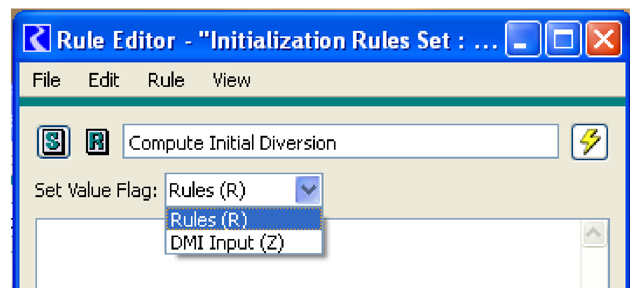
computational logic involved in this process, e.g., if the data need to be massaged to remove outliers, then this logic must somehow be embedded in the external DMI program. On the other hand, using initialization rules to accomplish the same task allows this logic to be viewed and edited from within RiverWare, often improving the clarity and ease of maintenance of the model.

Initialization Rules also differ from DMIs in that in the context of a RBS simulation, they can be used to provide default values which the policy can then override. To illustrate a scenario in which this might be useful, consider a run in which some quantity is generally assumed to be zero, but which can take on non-zero values under some conditions. One strategy for modeling this situation is to set the relevant values to zero using low priority rules with execution constraints that cause them to execute only on the first timestep. Alternatively, Initialization Rules accomplish this same behavior with less user effort (no execution constraints are necessary) and with the additional advantage that these rules are organized separately from the policy (if there is one). In fact, most RBS rules which are constrained to execute only on the first timestep are likely to be more appropriately included with the Initialization Rules.

Another important distinction between Initialization Rules and policy (rules) is that the former are executed before data checking at the beginning of the run. Thus, Initialization Rules can be used to set values such as initial Storage or Pool Elevation on a reservoir, whereas RBS rules execute only after data checking and so could not be used to give these slots reasonable values.

Behavior (How to use Initialization Rules): Following is a description of how to use initialization rules.

- **How do I access the Initialization Rule Set?** From the workspace, use the **Policy ➥ Initialization Rules RPL Set** menu.
- **Where are they saved?** The set is saved in the model file.
- **When are they executed?** The rules are executed during the initialization phase of all Simulation and Rulebased simulation runs (including accounting) after values are cleared and inputs are registered but before beginning of run checks on all objects. The exact ordering is described **HERE (Section 5)** for simulation, **HERE (RulebasedSimulation.pdf, Section 1.7.2)** for rulebased simulation, and **HERE (Accounting.pdf, Section 5)** for accounting.
- **In what order are they executed?** The Initialization Rules are executed in order of priority based on the specified “Agenda Order”. To see the order, click **View ➥ Show Advanced Properties.**
- **Do Initialization rules re-execute when dependent slots change?** No, initialization rules execute once and do not re-execute even if dependencies change.
- **What Flag/Priority are the values given?** Values set by Initialization Rules are given one of the following flags as configured by the user through the **Set Value Flag** menu on each Initialization Rule:



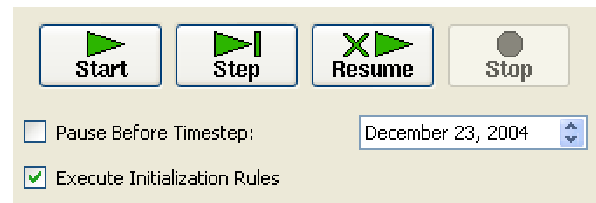
  - Rule (R) - Set the value with the “R” flag. In RBS, it is given a priority equal to one plus the number of rules in the RBS set. For example, if there are 27 rules in the RBS set, then values set by initialization rules would be given

priority 28. R flagged values set by Init rules can be overwritten like other R flagged values.

- DMI Input (Z) - Set the value with the “Z” flag and give it a priority of zero (0) (in RBS). When simulating, this flag behaves identically to the INPUT (I) flag. Note, an Initialization Rule setting a value with the Z flag can overwrite an “I” or “Z” flagged value.

Note: The values set by initialization rules are given the priority as described above. If these values are propagated or provide enough information for an object to dispatch, any subsequent values set will be given the controller priority, which would be zero (0).

- **Can I disable the Initialization Rules?** When the Initialization Rules set contains at least one rule and the current controller has the possibility of executing the Initialization Rules, then the Run Control Dialog displays a check box labelled "Execute Initialization Rules" which controls whether or not to execute the Initialization runs during the initialization phase of runs. Un-check this box to disable the Initialization Rules.



- **How do I debug Initialization Rules?** The RPL debugger can be used with Initialization Rules. Also, the following diagnostic categories in the Rulebased Simulation and Simulation settings dialog control diagnostics for the Initialization Rules:
  - **Init. Rules Print Statements**
  - **Init. Rules Rule Execution**
  - **Init. Rules Function Execution**

Debugging and analysis of RPL sets is described **HERE (RPLDebugging.pdf, Section 1)**.

5.2 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 **HERE (RunControl.pdf, Section 2.1)**. This allows models that lag or forecast to correctly compute values near the end of the run.

5.3 Under-determination

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.

6. 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.

6.1 2-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:

- The upstream Reservoir starts the iteration by dispatching with `solveMB_givenEnergyInflow` 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.
- The upstream and downstream Reservoirs iterate with the following steps until they converge:
    - The downstream Reservoir dispatches with `solveMB_givenInflowOutflow` 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.
    - The upstream Reservoir dispatches with `solveMB_givenEnergyInflow` 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.

6.2 Bi-Directional 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:

- One of the two Reservoirs dispatches with `solveMB_givenInflowOutflow`.
  - 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.
- The other Reservoir dispatches with `solveMB_givenInflowOutflow`.
  - 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.
- The Canal dispatches with `solveFlow`, which requires the following known values:
  - elevation1
    elevation2
    previousElevation1
    previousElevation2
  - 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 flow1 and flow2 slots. These values propagate across links back to the Reservoirs.
- Both Reservoirs re-dispatch independently. Since Canal Flow is linked and known, the dispatch can now solve for all of the Reservoir parameters.

See the documentation of the canal's dispatch method **HERE (Objects.pdf, Section 6.2.1)** for more information.

6.3 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:

- The Reach dispatches with one of two Dispatch Methods because of a known Inflow or Outflow.
  - If Inflow is known, the Dispatch Method is `solveOutflow`. If the Diversion slot on the Reach is linked but not known, the Available for Diversion slot is set equal to the Inflow.
  - If Outflow is known, the Dispatch Method is `solveInflow`. 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.
- The Available for Diversion value propagates across the link to the Aggregate Diversion Site.
- The Aggregate Diversion Site dispatches with processLumpedDiversion, processSequentialDiversion, 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.

- The Reach can now re-dispatch and solve for its Inflow or Outflow.