



Technical Documentation Version 5.1

Rulebased Simulation



C A D S W E S

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.

Rulebased Simulation Table of Contents

How Rulebased Simulation Works	1
Background: Review of Simulation	1
Rulebased Simulation	2
Rules and Rulesets	3
Rule Execution	3
Rule Dependencies	4
Rules Agenda	6
Rulebased Simulation Controller	7
Controller Priority	7
Simulation/Rules Interaction.....	7
Timestep Dependence of Rules.....	9
Slot Priorities and Flags	9
Resetting Slot Values	11
Series Slots.....	11
Multi Slots	13
Links.....	14
Dispatching	14
Governing Slots	15
Determination of Dispatch Method.....	17
Error Messages and Debugging	22
Errors and Ruleset Validity	22
Types of Errors.....	22
Ruleset Validation	23
Debugging Errors	24
Non-Runtime Errors.....	24
Non-Fatal Rule Evaluation Errors	26
Fatal Rule Evaluation Errors	28
Fatal Simulation Errors.....	31
Fatal Rulebased Simulation Errors.....	35
Debugging Tools	36

How Rulebased Simulation Works

1. How Rulebased Simulation Works

1.1 Background: Review of Simulation

Rulebased Simulation can only be understood once the fundamentals of basic Simulation have been mastered. This course assumes that you have a familiarity with simulation as presented in the RiverWare Introduction to Simulation Modeling course. 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.

Re-dispatching: 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 re-solve. In pure simulation, the object simply re-dispatches 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 re-set 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.

1.2 Rulebased Simulation

Rulebased Simulation is a variation of RiverWare's basic Simulation in which an under-determined model is provided additional information from user-specified rules which 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.

RULEBASED SIMULATION

Rulebased simulation is an extension of basic Simulation 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.

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.

1.3 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 slots in the model based on logic within the statement and possibly based on the values of slots in the model.

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:

```
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.

1.4 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.

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 two reasons: (1) the logic determines that no slot assignment is necessary because no value is returned by the RHS; or (2) 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 to #2, 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 three of these conditions are met: (1) all the information needed to evaluate all of the assignments is available; i.e., no NaNs are encountered in reference slots; (2) at least one of the assignments is effective (evaluates to a number); and (3) none of the assignments fails due to priorities.

1.5 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 assignment(s) — 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 re-fired.

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

Example:

Consider the following rule:

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

The first time the rule fires, the slot values are: SlotA = 30
SlotB = 20
SlotC = 10

The rule firing starts by clearing the dependency list.

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.

The rule assigns 30 to the SlotToBeAssigned, finishes successfully, and the dependency list is now: SlotA, SlotB and SlotC.

Now assume that during the ensuing dispatching, SlotB is recalculated to be 40, requiring that the rule fire again.

The second time the rule fires, the slot values are: SlotA = 30
SlotB = 40
SlotC = 10

The rule firing starts by clearing the dependency list.

The first IF evaluates to false, and both SlotA and SlotB are added to the dependency list.

There is no ELSE clause, so there is no RHS value. The rule finished ineffectually, and the dependency list is now: SlotA and SlotB.

Now assume that during the ensuing dispatch, SlotC is recalculated to be 50.

The rule is not fired again because SlotC is not a dependency; the result of firing the rule again would not be different.

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.

1.6 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:

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.

1.7 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.

1.7.1 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.

1.7.2 Simulation/Rules Interaction

The rulebased simulation controller behaves similarly to the simulation controller. (Details of the simulation controller are provided [HERE \(SimHowItWorks.pdf, Section 1\)](#).) 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 two steps:

1. Processing the object dispatch queue. Objects check their dispatch conditions and may try to dispatch (or re-dispatch) 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 dialogues. 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:

- Initialization and Beginning of Run
 - 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 and propagate user inputs across links for all timesteps.
 - Evaluate Beginning of run expression slots for all timesteps.
 - Execute Beginning of Run methods for all objects.
- For each timestep:
 - Set the controller clock to the timestep time.
 - Set controller priority to 0.
 - Execute Beginning of Timestep methods for all objects.
 - Evaluate expression slots that have evaluate-at-beginning-of-timestep selected

- Put all rules on the agenda (in priority order - whether 3,2,1 or 1,2,3 is user-selectable)
- Do the following two processes until both the dispatch queue and the rules agenda are empty
 - 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
 - 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.
 - Return to Process 1 and repeat until the Agenda and the Queue are empty.
- Set controller priority to zero, and execute End of Timestep methods on all objects.
- Evaluate End of timestep, current timestep only expression slots
- Execute End of Run methods on all objects.
- Evaluate End of run expression slots.

1.7.3 Timestep Dependence of Rules

In RiverWare's 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.

1.8 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** ➔ **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.

The 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 equivalency 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).

1.9 Resetting Slot Values

1.9.1 Series Slots

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 re-solve 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 may be made are the relative priorities of the existing slot value and the new slot value, and the presence (or lack of) **R** flags for the existing and new value. If a slot has an equivalent slot, the priority and **R** flag status of the equivalent slot is also checked. The determinations of whether or not to reset the slot value are as follows:

Rules for Overwriting an Existing Value in a Slot

User inputs (values which have an I flag) may never be overwritten.	
Never reset slot if:	old value flag* = I
Values which do not have an I flag are subject to the following conditions:	
Values which do not have an R flag may always be overwritten.	
Always Reset slot if:	old value rules flag* != R
Values which have an R flag may be overwritten by a value without an R flag if the priority of the new value is strictly higher (priority level is a smaller number) than the existing value's priority.	
Always Reset slot if:	old value rules flag* = R AND new value rules flag != R AND new value priority <i>higher than</i> old value priority
Values which have an R flag may be overwritten by a new value with an R flag if the new value's priority is equal to or higher than the existing value's priority	
Reset slot if:	old value rule flag* = R AND new value rule flag = R AND new value priority <i>higher than or equal to</i> old value priority

***Important 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.

Examples:

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

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

1.9.2 Multi Slots

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 below, 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.

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 re-calculate. That decision is based on which slot was solved for the last time and the priorities of each subslot:

3. 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.
4. 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.

*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:

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)

1.9.3 Links

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.

1.10 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 re-dispatch. But checking required knowns and unknowns is no longer a useful mechanism for determining the method with which to re-dispatch. In basic simulation, because slot values at a timestep are always set from the same source, the object simply re-dispatches 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 re-dispatches is the one whose required knowns have the highest priority slot values.

1.10.1 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.

Object	Potential Governing Slots
Aggregate Distribution Canal	Downstream Delivery Request
Aggregate Diversion Site	Total Available Water, Total Diversion Requested
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
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

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:

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:

solveMB_givenInflowStorage
and solveMB_givenInflowHW

The intersection of these dispatch methods' required knowns with the Storage Reservoir's potential governing slots yields the two following solutions:

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

1.10.2 Determination of Dispatch Method

As stated before, the determination of a dispatch method for re-dispatching must be 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 don't influence the choice of which dispatch method to solve with, there is no point in prioritizing them; all non-governing slots are added to this list first. Once this "minimum requirement" (the non-governing slots) for all dispatch methods has been met, the governing slots are added to the known list in priority order to determine the dispatch method. The algorithm is shown below:

Determination of the Correct Dispatch Method for Redischpatching

1. Begin with an empty set of known slots (pretend nothing is known).
2. Add to the known set all of the object's non-governing slots.
3. Add to the known set all of the object's potential governing slots with priority 0.
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.
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 the potential governing slots of all remaining priority levels until a dispatch method is determined. For each priority level:
If there are any slots at the priority level which have an **R** flag,
add the **R** flagged slots to the known set and defer the other slots.
If none of the slots at the priority level have an **R** flag,
add all of the slots at that priority to the known set.
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.
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.
5. Loop through all priority levels again, and for each:
Add any non **R**-flagged slots which were deferred in step 4 to the known set.
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.
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 loop.

This algorithm will always result in the identification of a dispatch method or an error and aborted run.

Example:

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 solveMB_givenInflowStorage 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	4 R
Hydrologic Inflow	0
Storage	3 R
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 redispach.

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:

Step 1):

Start with an empty known set.

Step 2):

Add all of the non-governing slots to the known set:.

Slot	Priority
Diversion	0
Return Flow	0

Step 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.

Step 4): 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 solveMB_givenOutflowStorage 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

Notice that 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 redispaches, 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 redispach 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.)

Error Messages and Debugging

2. Error Messages and Debugging

The following section describes debugging and error messages that are associated with Rulebased Simulation. For further information on diagnostics see the section [HERE \(Diagnostics.pdf, Section 3\)](#).

2.1 Errors and Ruleset Validity

2.1.1 Types of Errors

Several types of errors can be generated in RiverWare when preparing or running a Rulebased Simulation run. Depending on the type of error, the Rulebased Simulation Controller and the Rule Processor react differently. Some errors occur outside of a run, while others occur during the execution of a run. Some errors stop the run execution immediately, while others cause the current rule to end with an early termination, but do not stop the run. The approach to debugging these errors varies depending on the error type. The error types are:

Non-Runtime Errors: These are all of the errors which can occur when a run is not executing. These errors may be generated when loading a model, opening a ruleset, loading a ruleset, and/or editing a ruleset. When one of these errors is triggered, a message is generated in the Diagnostics Output Window or in a RiverWare confirmation dialog.

Runtime Errors: These are all of the errors which can occur while a run is executing. These errors may be generated within the Rulebased Simulation Controller, simulation dispatching, or the Rule Processor. Runtime errors are broken down into the following sub-types:

Non-Fatal Rule Evaluation Errors: These are errors which occur within the evaluation of a rule. Most often, these errors are produced when an engineering predefined function fails because it is attempting to model something physically impossible. These errors are often the result of the object state; i.e., the values in the object's slots. It is entirely possible for an engineering predefined function to fail during one rule evaluation, then succeed during another. The result of the engineering predefined function is almost always related to the state of the object at the time that the function evaluates.

When the Rule Processor encounters a non-fatal rule evaluation error, it cannot continue evaluating the rule and must abort it. Since the rule does not complete, no slots are set in the model. All existing slot values and priorities are still correct, and it is not necessary to stop the run. An error message is posted in the Diagnostics Output Window, and the Rule Processor fires the next rule on the agenda. The aborted rule may be put back on the agenda if any of its dependencies change.

Fatal Rule Evaluation Errors: These errors also occur within the evaluation of a rule. Unlike non-fatal rule evaluation errors, however, these errors are not dependant on the state of the system; they cannot be fixed by having different values in slots. These errors may be caused by missing arguments, arguments of the wrong data type, inconsistent unit types, invalid slot configurations, and/or missing objects and slots. These errors require intervention by the user if the run is ever to succeed. When the Rule Processor encounters a fatal rule evaluation error, the rule is aborted, the error message is posted to the Diagnostics Output Window, and the run is immediately stopped.

Fatal Simulation Errors: These errors occur during the dispatching phase of a Rulebased Simulation run. These errors are produced when the controller priority and the priorities of slot values are such that objects cannot solve the model. These errors are usually caused by objects dispatching with an unintended dispatch method, excessive redispaching of an object, and/or an overdetermined multislot. When these occur, the Rulebased Simulation Controller cannot determine how to continue, and the run is aborted with an error.

Fatal Rulebased Simulation Errors: These errors occur when a rule is fired too many times within any timestep. Rules may not be evaluated more than 50 times per timestep. A rule firing more than 50 times is usually an indicator of a circularity in the rule logic. Unchecked, circularities would continue indefinitely, or until the user terminates the executable. When a circularity is suspected, the Rulebased Simulation Controller aborts the run with an error indicating the name of the offending rule.

2.1.2 Ruleset Validation

Rulesets are validated at several stages prior to a run. Validation ensures that a ruleset meets a minimum level of “correctness” for its intended use. There are different levels of validity required for different stages of ruleset opening, editing, and running. The validity levels are:

Hopeless: A hopeless ruleset is one which cannot be read by the Rule Processor. This can occur if the ruleset file has been incorrectly modified with a text editor or if the file has been corrupted. In these cases, the Rule Processor cannot even read the ruleset to display it in the ruleset editor. If at any time, a ruleset is corrupted such that it cannot be opened, please contact RiverWare technical support.

Printable: A printable ruleset is one which can be read by the Rule Processor and displayed in the ruleset editor. This means that the expressions in rules and functions are syntactically correct, even though some may evaluate to data types which are inconsistent with their use in an outer scope. This level of validation is most frequently caused by an ambiguous expression being saved in the ruleset. Ambiguous expressions are expressions whose operator precedence allows more than one interpretation. Ambiguous expressions can be fixed by placing parentheses around the appropriate sub-expressions. A button for adding parentheses can be found in the rule palette.

When rulesets which only pass the printable validation are opened, errors are printed to the Diagnostics Output Window. The errors may then be fixed through the ruleset editor.

Consistent: A consistent ruleset is one which is printable and whose expressions' data types are consistent with each other. For example, a sub-expression of type NUMERIC exists where a NUMERIC expression is expected in a higher level expression. Some expressions may be unspecified if they are of the correct data type. The consistency of data types is determined for all types except LIST, whose member types cannot be fully determined until the expression is evaluated at run time. A consistent ruleset may not properly evaluate at run time, but it will not generate any errors when it is opened.

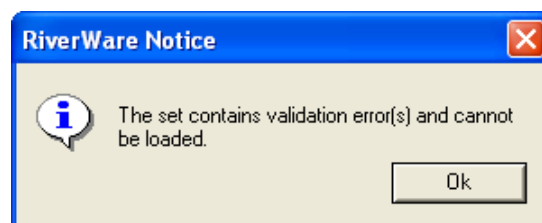
Evaluatable: An evaluatable ruleset is one which, as far as can be determined without actually evaluating it, could be successfully evaluated. This means that all of the previous validity levels have been satisfied. This also means that there cannot be any remaining unspecified expressions in the ruleset and that any object or slot references must map to an existing object or slot in the currently loaded model.

2.2 Debugging Errors

2.2.1 Non-Runtime Errors

There are several non-runtime errors which may be encountered when building or loading models and rulesets. These messages do not immediately affect a model run, but will likely result in a run failure if not addressed. Messages may appear in the Diagnostics Output Window or in an error notification window which appears at the time of the error.

Open Ruleset: When a ruleset is opened, its functions and rules are parsed from the ruleset file and the ruleset is validated to the printable level. The printable validation ensures that the ruleset can be displayed in the ruleset editor's graphical user interface. Rulesets are also validated to the same level when they are saved to ensure that they can be reloaded at a later time. Some older rulesets which contain critical errors, and corrupted ruleset files may fail to load. In these cases, a window appears indicating the location within the file where the error was detected:



A message will also appear in the Diagnostics Output Window indicating that the **Ruleset loading failed:** and provide the path and name of the ruleset. The Diagnostics Output Window will display the line number in which the parsing error occurred. If this should occur, first verify that the file you are attempting to load is indeed a ruleset, then contact RiverWare technical support.

Load Ruleset or Validate Ruleset: When a ruleset is loaded by clicking on the **Ruleset Not Loaded** button in the ruleset editor or **Check Validity** is selected from the **Ruleset** menu, the ruleset is validated

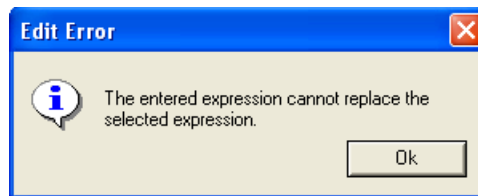
to the evaluable level. This means that all of the rule and function expressions are syntactically correct, have consistent expression data types, are fully specified, and reference objects and slots which exist on the workspace. If any of these criteria are not met, the ruleset is not validated and it cannot be loaded into the model. In this case, a window appears indicating one of the two messages below:



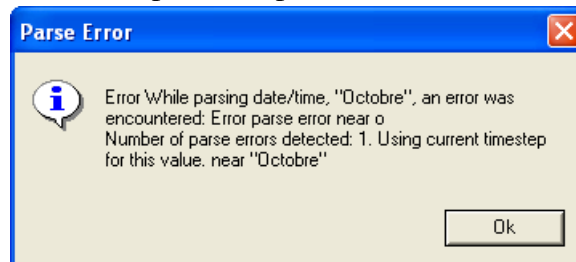
Another message appears in the Diagnostics Output Window describing the validation error(s).

Loaded rulesets are also validated to the evaluable level when a run is begun. This is to validate any changes which were made to the ruleset since it was successfully loaded. If any errors are detected, the run is stopped, and the appropriate diagnostic messages are displayed.

Expression Parsing: When an unspecified expression is filled in by typing text into its text field, the entry is parsed to ensure that it meets the requirements of the desired type. If the entry is not valid, a diagnostic message is generated. For example, typing @**Current Timestep** into an unspecified **<numeric expr>** would generate the following message and revert the expression to its prior, unspecified, state:



Another kind of parsing error can occur when the correct expression data type is improperly entered. For example, typing @**Octobre** into an unspecified **<datetime expr>** would generate the following message and revert the expression to its prior, unspecified, state:



In both of these cases, the entered expression was incorrect. Examination of the expression and the diagnostics message should point out the flaw.

2.2.2 Non-Fatal Rule Evaluation Errors

Non-fatal rule evaluation errors occur when a predefined function cannot evaluate given its current arguments or the current state of the system. Predefined functions which solve engineering algorithms can fail in this way if the arguments they are provided would result in violating a physical or data limitation of the object. For example, attempting to solve for the ending elevation of a reservoir given a starting elevation and flows which would cause overtopping of the reservoir would generate this kind of error. Some predefined functions can also fail in this way if they are attempting to access missing data or attempting to access data with incorrect arguments. Attempting to lookup values which do not exist in a tableslot would also generate a non-fatal evaluation error.

When a non-fatal evaluation error is encountered, the Rule Processor posts the error message to the Diagnostics Output Window and immediately aborts the rule. The rule's dependencies are preserved, such that the rule may be put back on the Agenda if any of its dependant slots change.

Example:

Consider the following rule which attempts to set Pool Elevation such that a target Storage is met. The **Target Storage()** is computed in an internal function. The **StorageToElevation()** predefined function is used to convert this target Storage to a Pool Elevation.

```
Res.Pool Elevation [] = StorageToElevation( %"Res", TargetStorage(),
    @"Current Timestep" )
```

If the **TargetStorage()** user function evaluated to a value of 15,000,000 acre-ft, and the ElevationVolumeTable of Res contains the following data:

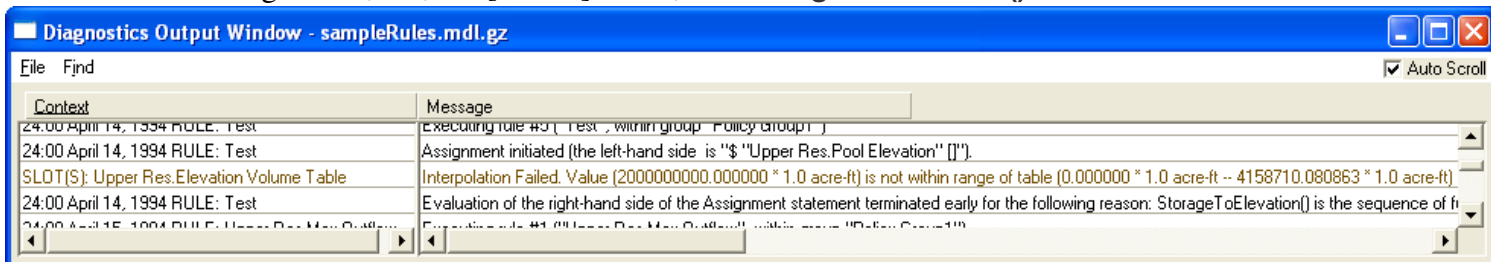
Pool Elevation	Storage
895.00	0
920.00	811,000
.	.
.	.
1060.00	8,241,000
1085.00	10,233,000
1110.00	12,452,000
1135.00	14,920,000

The predefined function, **StorageToElevation()**, will:

- Check the validity of the arguments:
Res is an object on the workspace which contains an Elevation Volume Table.
15,000,000 [acre-ft] is a numeric value in the unit type of volume.
@"Current Timestep" is a valid timestep of the current model run.
- Attempt to find 15,000,000 [acre-ft] in the Storage column of the table.
- Fail to find 15,000,000 [acre-ft] in the Storage column:

Post an error to the Diagnostics Output Window.
Notify the rule that the evaluation of the predefined function failed.

Notice that there is nothing wrong with the **StorageToElevation()** function itself. The arguments which were provided to it are hypothetically valid. It just happens that the Storage value is too large for the data in the table. This is not a ruleset configuration or logic error. It is possible that the same rule could execute at a later time and that the **TargetStorage()** function would solve for a Storage of 14,000,000 [acre-ft]. Then, the **StorageToElevation()** function would succeed.



When the rule is notified that the **StorageToElevation()** function has failed with a data error, it cannot finish evaluating. The rule is aborted because there is incomplete information to continue. Since the rule has not yet set any values in the model, the existing state of the model is still valid; it is based entirely on user input and assignments from successful rules. Since this failed rule has had no effect on the model, there is no need to stop the run. In addition, this rule's dependencies have been registered so that it can be re-fired if any of its dependant slots change at a later time. If Res' Storage or downstream demands were to change, this rule may re-fire and evaluate successfully.

Non-fatal rule evaluation errors do not always indicate a problem with a ruleset. Although a red error message is displayed in the Diagnostics Output Window, the failure of the rule may be a desired action. This may initially confuse users who are accustomed to seeing error messages only when the run has been aborted. Error messages which are generated for non-fatal rule evaluation errors are intended to give the modeler an indication of the failed functions even though the run continues. These error messages should be investigated to determine whether the function failure really was desired.

Example:

In the previous example, the rule was attempting to set Res' Pool Elevation corresponding to a target storage. If the Pool Elevation is higher than the reservoir capacity, it would not be wise to set this value on the slot. This would not be a valid reservoir operation, and the ensuing dispatch would surely fail, aborting the run.

If this rule is part of a ruleset where the Pool Elevation or Outflow of Res may be set to meet several criteria, we may want the rule to fail. Consider the following ruleset:

- Set Res Pool Elevation for Flood Control
- Set Res Pool Elevation for Target Storage
- Set Res Outflow for Surplus Conditions

Set Res Pool Elevation based on Guide Curve

If our Target Storage rule (#2) would result in a Pool Elevation which would overtop the reservoir, we may want to let the Surplus Conditions rule (#3) dictate the operation of the reservoir. Even though Rule #3 has a lower priority than Rule #2, we prefer its “safe” operation to Rule #2’s. By evaluating an engineering predefined function which will “test” the mass balance of the reservoir, we give Rule #2 a chance to fail before it sets any slots. Once Rule #2 fails with the non-fatal error, the Rule Processor fires Rule #3. The design of this ruleset ensures that an unrealistic operation will not be attempted and that the run will not be aborted.

When examining non-fatal error messages, keep in mind that the failing function does not know whether its failure is critical or not. In a simulation dispatching context, a failed Elevation Volume Table interpolation is critical because it means the physical limit of the reservoir has been exceeded. In a rule evaluation context, a failed Elevation Volume Table interpolation is not very critical because it only means that a “what-if” calculation has exceeded the physical limit. In both cases, the table interpolation error is posted as soon as it is encountered. In the simulation dispatch case, the controller should immediately stop the run. In the rule evaluation case, the Rule Processor only needs to abort this rule and can then fire the next rule on the agenda.

2.2.3 Fatal Rule Evaluation Errors

Fatal rule evaluation errors immediately abort the run. Unlike non-fatal rule evaluation errors, these errors indicate a major problem with the rule. These errors are not dependant on the state of the objects. Such an error in a rule would cause the rule to fail every time the rule is fired. The modeler must fix this error before a run can succeed.

Predefined Functions: The number of arguments and data types of arguments to predefined functions are fixed. Each function checks the validity of its arguments when it is evaluated. The argument requirements for each predefined function are listed in the *Rulebased Simulation - Predefined Function* section of the RiverWare help. If incorrect arguments are supplied to a predefined function, or the function fails to evaluate for other reasons, it may cause a fatal error. In this case, a message is posted in the Diagnostics Output Window and the run is immediately aborted.

Example:

The `ListSubbasin()` predefined function evaluates to a list of objects. The objects are the objects defined in the subbasin on the workspace. The function takes a single argument of type `STRING`. If the argument is not the name of an existing subbasin in the model, the function generates a fatal error.

Calling the function with a non-existent subbasin as below:

```
Print ListSubbasin ( "Mane Stem of River" )
```

The following error message is generated:

Evaluation of the Print statement failed for the following reason(s):

ListSubbasin() is the sequence of function calls (possibly containing only one function)

which ended in an unsuccessful function call.
 The function call failed for the following reason: Argument one is not a Subbasin.
 This occurred at the following location within the expression:
 “ListSubbasin (“Main Stem of River”)”.

Notice the structure of the error message. The first line indicates that the Print statement failed. The second and third lines indicate the reason the Print statement failed is that the evaluation of the **ListSubbasin()** function failed. The fourth line indicates that the function failed because the first argument is not a subbasin. The fifth and sixth lines indicate where in the rule the failed function call originated. Reading all of the error message is critical to understanding, and fixing, what went wrong.

Inconsistent Unit Types: The Rule Processor automatically converts numeric values to the proper units for mathematical operations during rule evaluation. As long as the unit type of a value is correct, its units can be converted without affecting the expression. If the dimensional analysis is incorrect, however, the Rule Processor stops the run.

Example:

The mathematical expressions below may be evaluated because the unit types of their elements are consistent.

Res.Outflow []= 10 [cms]+ 50 [cfs]
 FLOW= FLOW+ FLOW

Res.Storage []= 50,000 [acre-ft/month]x 31 [day]
 VOLUME= FLOWx TIME

The values of expression elements on the right-hand side are converted to a common base unit prior to evaluation. Likewise, the result of the right-hand side is converted into RiverWare standard units so that it may be assigned to the slot.

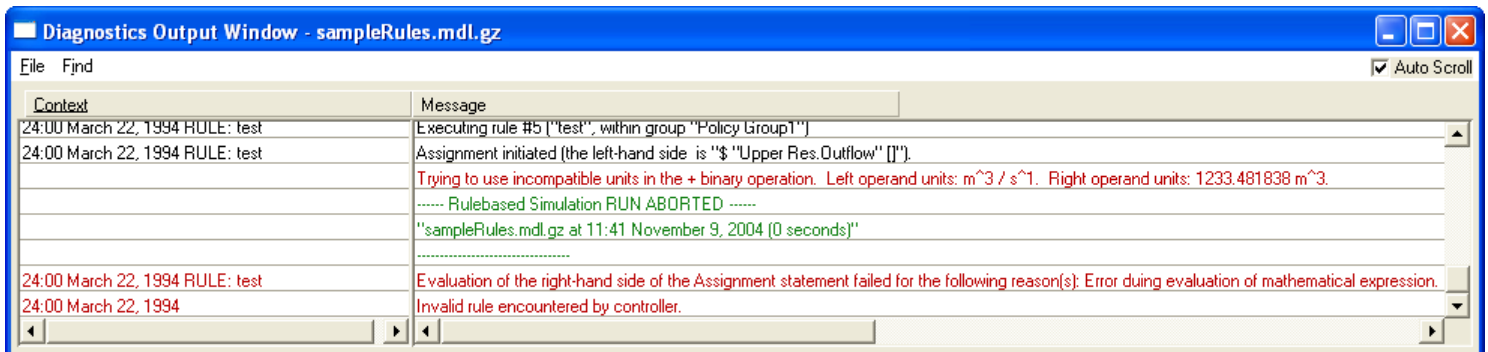
Example:

The mathematical expressions below may not be evaluated because the unit types of their elements are inconsistent.

Res.Outflow []= 10 [cms]+ 50 [acre-ft]
 FLOW= FLOW + VOLUME

Res.Pool Elevation []= 50,000 [acre-ft/month]x 31 [day]
 LENGTH= FLOW x TIME

These expressions would cause fatal rule evaluation errors like the one below.



Rule unit type errors are only detected during evaluation.

Inconsistent Data Types: The Rule Language requires that an expression evaluating to a particular data type be present where that data type is expected. This is enforced during the building of a rule by only enabling expressions of the correct type in the Palette and by generating a parse error if an incorrect expression is directly typed into the text field. The error message, previously discussed, would read: **“The entered expression cannot replace the selected expression.”**

Expressions of data type LIST are not verified during the building of a rule. This is because a list may contain elements of any data type and because the number and type of list elements may change during the evaluation of a rule. As a result, inconsistent data types originating in lists will not be caught until the rule or function is evaluated. These errors will stop the run immediately with an error.

Example:

The following rule will not generate any errors when it is constructed. When the ruleset is validated, it will be deemed “evaluatable.”

```
PRINT GET DATETIME @INDEX 0 FROM { 10 [cfs], @"January 15, 1999" }
```

When this rule is evaluated, however, it causes the run to abort with the following message:

Evaluation of the Print statement failed for the following reason(s):

Element number 0 of the list { 10.00000000 [cfs], @"24:00:00 January 15, 1999" }
is of type NUMERIC, whereas it should be of type DATETIME.

This occurred at the following location within the expression:

```
GET DATETIME @INDEX 0 FROM { 10.00000000 ["cfs"], 24:00 January 15, 1999 }
```

Here again, the message indicates exactly where in the rule the error occurred and what went wrong.

Missing Objects or Slots: If an object or slot which is referenced in a rule does not exist on the workspace when the rule is evaluated, a fatal rule evaluation is produced. This can occur when the slot or object name was improperly entered in the rule, when a ruleset is being used with the wrong model, or when object or data object slot names were changed after the rule was created.

Names of objects and slots in a ruleset must exactly match an object and/or slot name on the workspace, including capitalization, spaces, and any underscores. Missing object or slot errors can be minimized by always using the Object Selector and Slot Selector to specify objects and slots in rules and functions. Typing object or slot names directly into an expression textfield is not recommended due to the possibility of a syntax error.

Example:

The following rule will not generate any errors when it is constructed. When the ruleset is validated, it will be deemed “evaluatable.”

```
Reservoir.Outflow[] = Reservoir.Inflow[]
```

When this rule is evaluated, however, it causes the run to abort with the following message:

Evaluation of the right-hand side of the Assignment statement failed for the following reason(s):

Failed to locate slot “Resarvoir.Inflow” on the global workspace.

This occurred at the following location within the expression: \$”Resarvoir.Inflow”.

2.2.4 Fatal Simulation Errors

Fatal Simulation errors are often the most complex errors to understand and debug. These errors occur during the simulation dispatching which takes place after a rule successfully sets a slot value. In order to simulate the effects of the new slot value, many objects may need to redispach. The dispatch method with which each object redispaches is determined from the object’s slot priorities. Occasionally, the priorities of the slots are such that a unique dispatch method cannot be identified. When this happens, the object will attempt to redispach with the same method as it dispatched with the last time. One of two error situations may result.

Junior/Senior Slot Priority Error: If the slot which was just set by a rule is solved for during this dispatch, the slot priority conflict will cause the run to abort with an error. The error often indicates:

Attempting to set senior slot (*priority*) with a junior priority (*same priority*).

Assignment attempted within the dispatch triggered by rule #*same priority*.

The object probably did not dispatch with the intended method.

When this error occurs, the rule at the indicated priority is usually to blame. Often, this rule has set a slot value at a priority which confuses the simulation into choosing the wrong dispatch method.

Example:

Consider a Reach object called Rio. Rio’s Inflow may be determined by the release of an upstream reservoir or its Outflow may be determined by downstream demands or environmental considerations. Both may be specified as long as the slot’s priorities can be used to determine the preferred solution.

The ruleset used to control Rio is shown below:

1. Set Rio Inflow due to Minimum Upstream Reservoir Release
**Rio.Inflow[] = IF (Rio.Inflow[] < DataObj.MinResRelease[]) THEN
DataObj.MinResRelease[]**
2. Set Rio Outflow for Maximum Fish Flow
**Rio.Outflow[] = IF (Rio.Outflow[] > DataObj.MaxFishFlow[]) THEN
DataObj.MaxFishFlow[]**
3. Set Rio Outflow to Meet Downstream Demands
Rio.Outflow[] = DataObj.TotalDownstreamDemand[]

The Rulebased Simulation proceeds as follows:

- Initially, no values are known, so Rio cannot dispatch.
- Rule #1 fires but terminates early because Rio.Inflow is unknown.
- Rule #2 fires but terminates early because Rio.Outflow is unknown.
- Rule #3 fires and sets Rio.Outflow to meet demands.
- Rio dispatches with the SolveInflow dispatch method based on these priorities:
Inflow = unknown
Outflow = known at priority 3R
and solves for Inflow at controller priority 3. This puts rules #1 and #2 back onto the Agenda.
- Rule #1 fires and sets Rio.Inflow for minimum upstream release (assuming the rule logic evaluates to TRUE). The rule can overwrite the existing priority 3 Inflow with the new priority 1R Inflow.
- Rio redispaches with the SolveOutflow dispatch method based on these priorities:
Inflow = known at priority 1R
Outflow = known at priority 3R
and solves for Outflow at controller priority 1. Rule #2 is still on the Agenda.
- ↳ Rule #2 fires and sets Rio.Outflow for maximum fish flow (assuming the rule logic evaluates to TRUE). The rule can overwrite the existing priority 1 Outflow with the new priority 2R Outflow.
- ↳ Rio redispaches with the SolveOutflow dispatch method based on these priorities:
Inflow = known at priority 1R
Outflow = known at priority 2R
and solves for Outflow at controller priority 2. This was not the intended dispatch! Rule #2 just set Rio.Outflow. The effect of this rule should not be to solve for a new Rio.Outflow. When the dispatch attempts to overwrite Outflow at priority 2R with a new value at priority 2, an error is generated.

The responsibility for the error, in this case, falls squarely on Rule #2. This rule should not be attempting to enforce a maximum fish flow when a higher priority minimum release rule is controlling the river. Unfortunately, Rule #2 does not know that Rule #1 is controlling the river. There are two solutions to this error.

Because the rules are acting on different slots, they cannot depend exclusively on priorities to determine the appropriate solution. If Rule #2 were attempting to set Rio.Inflow instead, the rule would fail gracefully during the rule execution. The existing priority 1R would prevent Rule #2 from

overwriting the slot value. Of course, changing Rule #2 to set Inflow could now create a similar problem between Rule #2 and Rule #3.

The other solution is to make Rule #2 “smarter.” Rule #2 could check the Inflow of Rio to see if it is at a level corresponding to the minimum upstream release. If this is the case, Rule #2 could then decide not to return a value from its right-hand side. It would exit ineffectually. Another way to let Rule #2 know that Rule #1 is controlling the river is to have Rule #1 set a “state flag” on a data object in the model. State flags are slots whose values indicate the current state of the system, such as surplus, shortage, or minimum release. A state flag could be checked by Rule #2 to decide if it should change Rio’s Outflow.

Infinite Loop Dispatching Error: Another error situation which can result from incorrect dispatching is an infinite loop. This can occur between two objects which are both solving for the same slot. Each object successfully solves for a new value on the slot, overwriting the previous value. Each new slot assignment triggers the other object to redispach, during which it also solves for a new value on the slot. This continues until the maximum iterations are reached on one of the objects’ slots or the modeler terminates the RiverWare session.

Example:

Consider two Reach objects called UpperRio and LowerRio and a Diversion called UpperRioDiversion. The diversion is attached to the UpperRio Reach and can divert an amount of flow equal to the amount it leaves in the UpperRio. There are also irrigation demands downstream of the LowerRio. The system usually solves from the bottom up; downstream demands are set on LowerRio’s Outflow, then the Diversion Request is set on the UpperRioDiversion. As in the previous example, UpperRio’s Inflow may be overwritten by a minimum release rule.

The ruleset used to control the Rios is shown below:

1. Set UpperRio Inflow due to Minimum Upstream Reservoir Release
**UpperRio.Inflow[] = IF (UpperRio.Inflow[] < DataObj.MinResRelease[]) THEN
 DataObj.MinResRelease[]**
2. Set UpperRioDiversion Not to Exceed UpperRio.Outflow
**UpperRioDiversion.Diversion Request[] = MIN (DataObj.UpperRioDivRequest[],
 UpperRio.Outflow[])**
3. Set LowerRio Outflow to Meet Downstream Demands
LowerRio.Outflow[] = DataObj.TotalDownstreamDemand[]

The Rulebased Simulation proceeds as follows:

- Initially, no values are known, so neither Rio can dispatch.
- Rule #1 fires but terminates early because UpperRio.Inflow is unknown.
- Rule #2 fires but terminates early because UpperRio.Outflow is unknown.
- Rule #3 fires and sets LowerRio.Outflow to meet demands.
- LowerRio dispatches with the SolveInflow dispatch method based on these

priorities:

Inflow = unknown

Outflow = known at priority 3R

and solves for Inflow at controller priority 3. This propagates across the link to UpperRio.Outflow, causing Rule #2 to be put back onto the Agenda.

- UpperRio dispatches with the SolveInflow dispatch method based on these priorities:
Inflow = unknown
Outflow = known at priority 3
but cannot solve completely because the UpperRioDiversions is not yet known.
- Rule #2 fires and sets the UpperRioDiversions's Diversion Request equal to the UpperRio Outflow (assuming that the request in the Data Obj is very large).
- UpperRioDiversions dispatches and sets the Diversion from UpperRio at controller priority 2. This propagates across the link to UpperRio.Diversion.
- UpperRio redispaches with the SolveInflow dispatch method based on these priorities:
Inflow = unknown
Outflow = known at priority 3
Diversion = known at priority 2
and solves for UpperRio.Inflow at controller priority 2. This causes Rule #1 to be put back onto the Agenda.
- Rule #1 fires and sets UpperRio.Inflow for minimum upstream release (assuming the rule logic evaluates to TRUE). The rule can overwrite the existing priority 2 Inflow with the new priority 1R Inflow.
- UpperRio redispaches with the SolveOutflow dispatch method based on these priorities:
Inflow = known at priority 1R
Outflow = known at priority 3
Diversion = known at priority 2
and solves for UpperRio.Outflow at controller priority 1. This puts Rule #2 back on the Agenda.
- LowerRio redispaches with the SolveOutflow dispatch method based on these priorities:
Inflow = known at priority 1
Outflow = known at priority 3R
and solves for Outflow at controller priority 1.

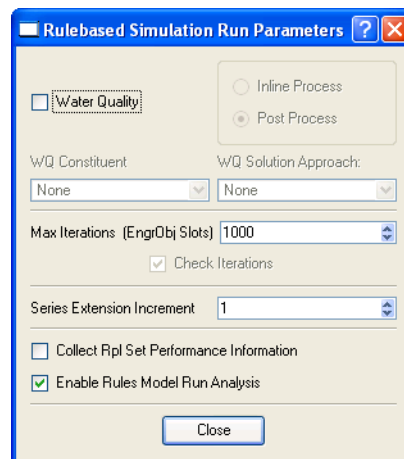
Now the stage is set for the error to take over.

- Rule #2 fires and resets the UpperRioDiversions's Diversion Request equal to the new UpperRio Outflow (assuming, again, that the request in the Data Obj is very large). The rule can overwrite the existing priority 2R Diversion Request with a new priority 2R Diversion Request.
- UpperRio redispaches with the SolveOutflow dispatch method based on these priorities:
Inflow = known at priority 1R
Outflow = known at priority 3
Diversion = known at priority 2
and solves for a new UpperRio.Outflow at controller priority 2. The dispatch can overwrite the existing priority 1 Outflow with a new priority 2 Outflow. This value propagates across the link to

LowerRio's Outflow. This also causes Rule #2 to be put back on the Agenda.

- LowerRio redispaches with the SolveInflow dispatch method based on these priorities:
Inflow = known at priority 2
Outflow = known at priority 1
and solves for a new Inflow at controller priority 2. This propagates across the link to UpperRio.Outflow.
- UpperRio redispaches again with the SolveOutflow dispatch method based on these priorities:
Inflow = known at priority 1R
Outflow = known at priority 2
Diversion = known at priority 2
and solves for a new UpperRio.Outflow at controller priority 2. This value propagates across the link to LowerRio's Outflow.
- The two Rios continue redispaching with the wrong dispatch methods and overwriting the results of each other's solutions.

Slot maximum iteration checking is turned on by default and cannot be turned off when performing Rulebased Simulation, though the number of maximum iterations can be changed by the user. This iteration default can be viewed as a grayed box for "Check Iterations" in the **Rulebased Simulation Run Parameters** dialog, invoked from the **View** menu of the **Run Control** dialog. Within this dialog, the user can change the number of maximum iterations (the default number of maximum iterations is 20). For more complex models, the user may be required to increase the number of maximum iterations.



2.2.5 Fatal Rulebased Simulation Errors

Fatal Rulebased Simulation errors are relatively rare. The only error of this type occurs when a rule is fired more than 50 times in a single timestep. Even in the most complex rulesets, individual rules should not fire more than a few times in any given timestep. Excessive redispaching is an indication of a circularity in the ruleset's design; one or more rules are dependent on the slots which they are setting. When a rule circularity is suspected, the Rulebased Simulation Controller stops the run and posts this error to the Diagnostics Output Window:

Max rule executions exceeded for timestep.

Example:

Circularities in rulesets are easy to create consciously. Luckily, they rarely happen by accident.

An example of a circular ruleset which causes no dispatching, is shown below:

```
.Data.A[] = IF ( Data.B[] == 1 [NONE] ) THEN
  0 [NONE]
  ELSE
  1 [NONE]

Data.B[] = IF ( Data.A[] == 1 [NONE] ) THEN
  1 [NONE]
  ELSE
  0 [NONE]

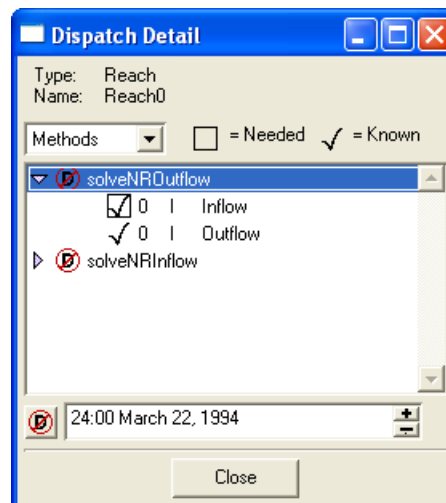
Data.A[] = 1 [NONE]
```

The Rulebased Simulation would proceed as follows:

- Rule #1 fires but terminates early because Data.B is not known.
- Rule #2 fires but terminates early because Data.A is not known.
- Rule #3 fires and sets Data.A to 1. Rule #2 goes back on the Agenda.
- Rule #2 fires and sets Data.B to 1. Rule #1 goes back on the Agenda.
- Rule #1 fires and sets Data.A to 0. Rule #2 goes back on the Agenda.
- Rule #2 fires and sets Data.B to 0. Rule #1 goes back on the Agenda.
- Rule #1 fires and sets Data.A to 1. Rule #2 goes back on the Agenda.
- The steps above are repeated another 24 times before the Rulebased Simulation Controller stops the run.

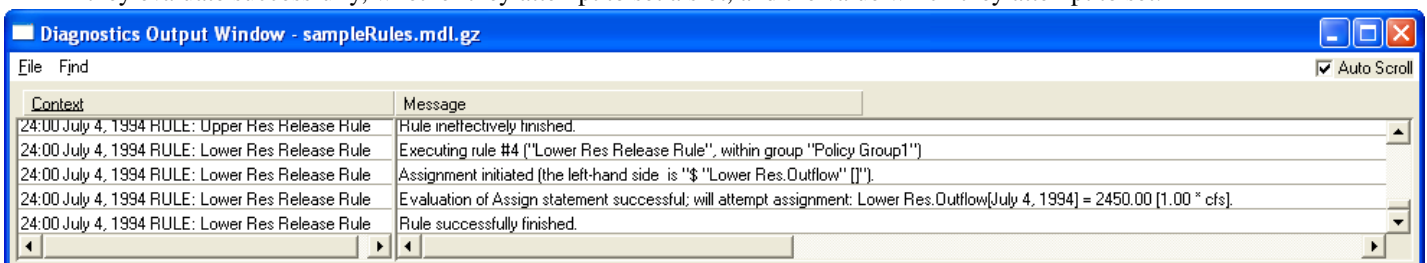
2.3 Debugging Tools

The Rulebased Simulation Analysis Dialog is the best tool for identifying the source of fatal runtime errors. The Rulebased Simulation Analysis Dialog provides a snapshot of the last known state of the objects at each timestep. Its detail dialog shows the priorities of each dispatching slot and the last dispatch method used. This information can be used to identify priority conflicts which are the source of many errors.

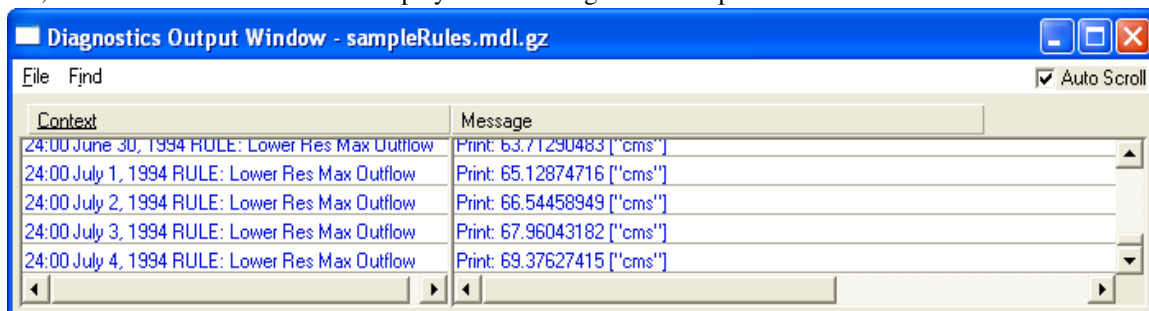


When intermediate information is required to debug a simulation error, the Rulebased Simulation Diagnostics should be used. Some diagnostics groups are more useful in debugging errors than others. The diagnostics groups are listed below, ordered from the most frequently used to the least frequently used. Diagnostics groups which are not described below provide the same information in Rulebased Simulation as in Simulation.

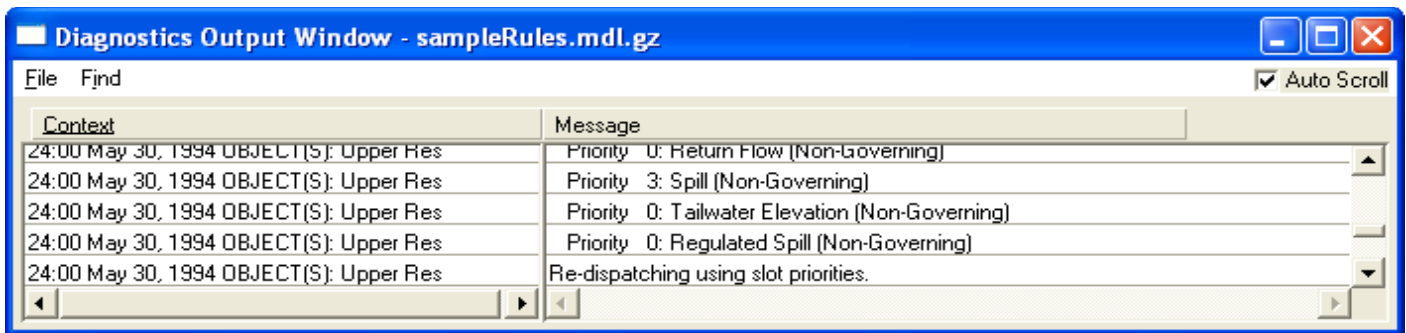
- The **Rule Execution** diagnostics group provides information about the evaluation of rules, including when they fire, whether they evaluate successfully, whether they attempt to set a slot, and the value which they attempt to set.



- The **Print Statements** diagnostics group enables the printing of PRINT statements in rules. If this diagnostics group is not turned on, no PRINT statements will be displayed in the Diagnostics Output Window.



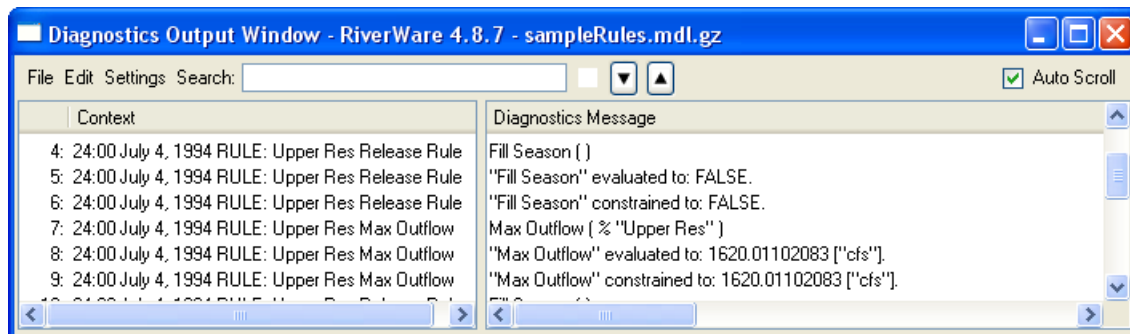
- The **Dispatch Management -> SimObj** diagnostics group provides information about the priorities of slots to determine the dispatch method for each object. This is especially critical when evaluating whether an object dispatched with the intended method.



- The **Dispatch Management** -> **Controller** diagnostics group generates a message whenever an object begins dispatching and provides the controller priority during the dispatch.



- The **Function Execution** diagnostics group provides information on the execution of predefined and user defined functions. When this group is enabled, function diagnostics are posted for all functions that have "Before Execution" and/or "After Execution" diagnostics enabled. "Before Execution" diagnostics show when the function is called and provides the arguments passed into the function. "After Execution" diagnostics show the result from the function call and any post-execution constraints imposed.



- The **Rule Management** -> **Dependencies** diagnostics group provides information about the dependencies of the current rule being evaluated. After the rule execution is complete (successful or not), this diagnostics group posts a list of all slots which are dependencies for this rule.



- The **Rule Management** -> **Agenda** diagnostics group provides information about the state of the Agenda. After each rule executes, this diagnostics group posts a list of all rules on the Agenda. The diagnostics group also posts a message whenever a rule is placed back on the Agenda due to a change of one of its dependencies.



- The **Rule Management** -> **Cache** diagnostics group provides information about all of the slots which a rule is attempting to set. All slots are set as a group only when the rule completes successfully and all of the slot assignments are verified for priority and maximum iterations. This diagnostic group prints the result of the attempted assignments.

