DESIGN PROCESS FOR APPLICATION-SPECIFIC LANGUAGES: A LANGUAGE FOR WATER RESOURCES POLICY SPECIFICATION

by

STEPHEN C. WEHREND

B.A., University of Colorado, 1984

M.S., University of Colorado, 1990

Application-specific languages are custom programming languages that are used within the context of a single application. They provide users with functionality that cannot be readily built into an application through other means, such as direct manipulation interfaces. Generally, application-specific languages are appropriate where the problems to be solved are not completely known in advance and require the use of complex logic or flow of control.

Application-specific languages are frequently designed and developed, yet there is little literature published on their design. What is published tends to focus on the domain of the problem for which the language is needed, the language itself and how well the language addresses the problems for which it was created. There is little discussion regarding what design decisions were made and why, what alternatives were considered and what components might need to be a part of an application-specific language system.

In an effort to shed light on the design of application-specific languages and to provide guidance for future designers, this dissertation presents a multi-part case study in which two application-specific languages were designed and developed. The language is used to specify policies that are part of water resources simulations. The target application, *RiverWare*, is a general purpose river basin modeling tool. The policies specified in the language are used by *RiverWare's* simulation engine to alter the flow of water in the river basin. The users of this system and language are water resources engineers whose programming experience and abilities range from minimal to moderate.

Acknowledgments

I would like to thank the following:

- my advisor, Clayton Lewis, for his guidance, insight and patience during this research;
- René Reitsma, for his assistance during this research and his input on the design of the language;
- Terry Fulp, for his sponsorship of the language and his input throughout the process;
- Liz Jessup and Amer Diwan, for their help in guiding this research;
- Edie Zagona, for her leadership of the *RiverWare* project and her support and input;
- Brad Vickers and Bruce Williams, for providing target problems and input that were critical to the design of the language;
- Patrick Lynn and Alan Eliasen, for their assistance with the development of the second language and their insights and feedback on its design;
- Bill Oakley, for the significant contributions he has made to *RiverWare*; and
- Jennifer, Jonathan and Benjamin, for their patience, support and encouragement throughout the entire process.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1. Application-specific Languages

Application-specific languages are custom languages that are intended to be used within the context of a single application. They are often smaller and less powerful than their general purpose counterparts and are equipped with features that allow them to interact with their host application. These features can allow the programs written in the language to exchange information with the host application, provide the host application with the ability to control the execution of the programs and allow the programs to request the execution of application-specific functionality. In addition, an application-specific language may provide a programming environment to support the creation, modification and debugging of the programs.

The functionality provided by application-specific languages is usually complex enough to require some level of programming that cannot easily be accomplished by other methods, such as direct manipulation interfaces. Indeed, most application-specific languages are included because the range of problems that must be solved by the application defy any predefined solution and require the use of a more general purpose approach. As with any language, an application-specific language is intended to express complex logical and relational operations and can also include various control structures.

Application-specific languages can be found within many applications ranging from widely used, commercial applications to more limited use, domain-specific applications. Commercial applications include spreadsheets, such as *Microsoft Excel*, which contains a macro language that is used for executing complex equations and logic within or between cells [Roman, 1999]. They also include technical computing programs, such as *Mathematica*, which relies on an application-specific

language to allow users to represent complex mathematical expressions [Wolfram, 2001]. In both cases, syntax and semantics match the tasks for which they are intended.

Domain-specific applications often contain languages that are used to extend the capabilities of the application in any number of ways. *RiverWare*, the application that was used as the basis for this research, uses ten application-specific languages. These include:

- a model specification language,
- a batch mode execution language,
- a language used to specify arithmetic expressions,
- a language used to specify model-specific user preferences,
- a data import/export language,
- a language used to support optimization,
- a unit specification language,
- a date/time specification language,
- a language used to support a spreadsheet style model viewer and
- a rule-based simulation policy language.

1.2. General Purpose and Domain-specific Languages

Application-specific languages are both similar to and different from general purpose and domain-specific languages. Whereas all of these language types can be used to solve problems using some degree of complex logic and control structures, they differ in some fundamental respects. Application-specific languages are intended to be used within a single application and address a particular range of problems within that application. General purpose languages, such as C, Pascal or LISP, are not designed to apply to any specific problem or type of problem, but are intended to be used to solve a wide range of problems. Although these languages have their individual strengths and weaknesses when it comes to solving certain types of problems, they generally can be used interchangeably. For someone wishing to design a new general purpose language, the established literature details the design of these languages [e.g., Arnold, Gosling and Holmes, 2000, Steele, 1994,

Hudak and Fasel, 1992, Mössenböck and Wirth, 1991, Unger and Smith, 1991, Harbison, 1990, Mössenböck and Templ, 1989, Holt, et. al., 1988, Wirth, 1984, Pratt, 1984, Harland, 1984, Hilfinger, 1983, Smedema, Medema and Boasson, 1983, Horowitz, 1983, MacLennan, 1983, Hoare, 1980, Wasserman, 1980, Williams and Fisher, 1977, Wirth, 1974].

Domain-specific languages are designed to be applied to a particular domain or problem type [Spinellis and Guruprasad, 1997, van Deursen and Klint, 1997, Chandra, Richards and Larus, 1996, Derby, Schnabel and Zorn, 1995, Buckley and Wheatcraft, 1991]. While this is often true of application-specific languages, domain-specific languages differ in that they are intended to be used outside the context of any particular application. An example of a domain-specific language is AMPL, which is designed "to help people use computers to develop and apply mathematical programming models" [Fourer, Gay and Kernighan, 1993]. Unlike a general purpose language, AMPL is to be used to solve a limited set of problems and, unlike an application-specific language, AMPL is to be used outside the context of a single application.

1.3. Thesis

Given the frequency with which application-specific languages are designed and developed and the number of application-specific languages that are in existence, one would expect there to be a well developed body of literature on their design. Unfortunately little literature on the design of these languages is available. What is published tends to consist of a description of the domain of the problem for which the language is needed, a description of the language itself and a discussion about how well the language addresses the problems for which it was created [e.g., Kohler, Poletto and Montgomery, 1999, Crivelli, 1995, Williams and Hann, 1972]. There are rarely discussions about what design decisions were made and why, what alternatives were considered and what components might need to be a part of an application-specific language system.

In light of the lack of literature on the design of application-specific languages, I have conducted a multi-part case study in which I designed and implemented a two generation application-

specific language. The language is used to specify policies that are utilized by water resources simulation models. The target application, *RiverWare*, is a general purpose river basin modeling tool [Zagona, et al., 2001, Magee, Zagona and Frevert 2001]. The policies specified in the language are used by *RiverWare's* simulation engine to allocate water within a river basin over a predefined time range. The users of this system and language are water resources engineers whose programming experience and abilities range from minimal to moderate.

As a result of these case studies, a variety of insights into the design of these languages and many of the issues that are likely to confront future designers of application-specific languages were gained. Whereas these insights will not apply to all application-specific language design problems, they should provide a basis for those who need to design application-specific languages and for those who intend to delve more deeply into the process of application-specific language design.

1.4. Methodology

A case study was selected as a means to both gain insight into the issues and processes involved in designing application-specific languages and to allow for the design and development of a set of application-specific languages to be used within *RiverWare*. A case study was deemed appropriate given that a real application-specific language needed to be developed for a group of users who would use the language as part of their job. A subset of these users would also be available to assist in the development of problems that would have to be programmed using the language. In addition, once the language was finished, all of these users would be able to provide feedback on how well the language met their needs.

A case study also seemed appropriate given that the task of designing and implementing this language was essentially exploratory in nature [Yin, 1994]. In addition, since this area is not well-researched and thus there is little literature on the subject [Leonard-Barton, 1995], a case study would provide a means of gathering information on the process of application-specific language design and allow for the inclusion of real-life constraints on the problem [Buckley, Buckley and Chiang, 1976]. The

inclusion of real-life constraints seems particularly important given that these languages may be developed by those who are not trained in language design and there are many factors that must be taken into account that might not become apparent if the design process is studied outside the context of an actual problem. These factors include time, budget and personnel constraints. Lastly, the language developed needed to be used within a commercially available application. This gave the research a significant practical aspect.

Although a case study was deemed to be the best approach, there are pitfalls in doing case study research. The main drawback being that the results produced do not necessarily generalize [Buckley, Buckley and Chiang, 1976, Eisenhardt, 1995]. Accordingly, the results from these studies may not be applicable to the design of other application-specific languages. To a degree, this was partially mitigated by performing two case studies, one for each language. The additional insights gained by the inclusion of the second case study will augment the generalizability of the research [Leonard-Barton, 1995].

Chapter 2

Background for Application-specific Languages in Water Resources

The focus of this study is the design and implementation of an application-specific programming language for the specification of policy within a water resources modeling system. The application included a simulation component that modeled the flow of water through a river basin over a user-defined range of time. The application-specific language was needed to allow the users to specify policy rules that altered the flow of water through the river basin. These policies were extracted from an existing body of law and were used to insure that the simulation modeled the flow of water so that it could be allocated in accordance with the law.

The following sections discuss some of the background information that relates to this study. This information includes the problem domain, the people involved, how the language was to be used and an overview of the application within which the language would be used. The chapter concludes with a brief discussion of the motivation for the creation of the policy languages that were the subject of the studies.

2.1. CADSWES

The studies were performed at the Center for Advanced Decision Support for Water and Environmental Systems (CADSWES). CADSWES is an interdisciplinary center that is involved in the research, design and implementation of environmental decision support systems. It is part of the Department of Civil, Environmental, and Architectural Engineering, of the College of Engineering and Applied Science at the University of Colorado at Boulder.

CADSWES works with sponsor organizations such as the United States Bureau of Reclamation (USBR) and the Tennessee Valley Authority (TVA) to design and develop data-centered, graphically-based decision support systems. These systems are used to model and provide information about the water systems that these organizations manage. The systems are modeled so that the organizations can schedule, forecast and plan the use of many of the resources contained within these physical systems. CADSWES employs civil engineers, computer scientists and social scientists who work together to design and build these systems.

2.2. Domain

The domain of this study is water resources engineering. Water resources engineering is a broad discipline that encompasses many areas related to the allocation, control and management of water systems. These water systems can be both above and below ground, although for the purposes of this study, only above ground, or surface water, systems are considered. Surface water, in turn, can be further categorized into areas related to water quantity and water quality. The following sections will provide an overview of water systems, water quantity and water quality.

2.2.1. River Systems

River systems are networks of surface water that are contained within one or more river basins. A river basin, in turn, is the land area that drains into a river. As an example, suppose a river runs through a valley. The river basin within which this river runs includes the river and those portions of the surrounding hills that drain water into the river. Those portions of the hills that drain water into another river are part of another river basin. Smaller river basins can be combined form larger river basins. This is true when small rivers meet and form larger rivers.

River basins are composed of a number of physical entities such as rivers, canals, lakes, reservoirs, confluences and diversions. While river basins also include the surrounding land, these areas are generally considered relevant only as sources and destinations of the water contained by the physical entities. Although many of these terms are known by most people, some may be new. In addition, their

use within the water resources community may differ from the colloquial use. Accordingly, a brief description of each of these entities follows for completeness.

River. A river is a natural waterway wherein water travels downstream to some other river basin entity, such as a reservoir. Stretches of rivers between river basin entities are often referred to as *reaches*. With the exception of abnormal situations, such as earthquakes, water in a river reach flows downstream only. In addition, there is generally a maximum flow, or volume per time unit, which can be accommodated within a river reach.

Canal. A canal is a man-made waterway that serves much the same purpose as and has many of the same characteristics of a river. Canals differ, however, in that they can be designed to allow water to flow in either direction.

Lake. A lake is a natural body of water that generally formed as a result of rainfall or snow melt runoff or as the termination point of a river reach or canal. For this study, lakes are considered special cases of reservoirs, which are described below. More to the point, lakes are just reservoirs in which the outflow, if there is any, is equal to the inflow, minus evaporation and any other loss.

Reservoir. A reservoir is a man-made lake wherein a river is dammed and water is backed up upstream of it. Like lakes, reservoirs cannot control the amount of water that flows into them. Unlike lakes, however, reservoirs can control the amount of water that flows out of them. This relatively simple feature gives reservoirs the ability to store water, control floods, maintain a consistent outflow of water, provide a recreational environment and allow for hydropower generation.

Confluence. A confluence is a place where river reaches and/or canals join to form a single river reach or canal. This resulting reach or canal contains the combined water of the two sources. For modeling purposes, runoff from farms, rain-fall and snow-melt are generally modeled using one or more confluences.

Diversion. A diversion is a place where reaches or canals split. Sometimes this results in two reaches, two canals or a reach and a canal where there had formerly had been one. Often, however, a diversion is used to model a location where water is being taken from the system. For instance, when a canal passes a farm that uses some of the water in the canal, a diversion is used to take the water from the system. In these cases, the reach or canal 'deposits' some water and continues on. As suggested above, some of this water may later return to the river system as a result of runoff.

2.2.2. Water Quantity

Water quantity is concerned with the amount and distribution of water in the river system. There must be sufficient water in the system in order to provide for those who use the water. Some typical uses include irrigation and drinking, which are consumptive uses, and recreation and hydropower, which are non-consumptive uses. In addition, enough water must be stored in the system to provide reserves that might be needed in times of future shortages.

The system must also have the water distributed such that it can accommodate the needs and prescribed allocations of the users. For instance, sufficient water must be stored in reservoirs to allow for the maximum generation of hydropower without compromising other needs, such as safety or irrigation. Water must also be available throughout the system so that the water users can get water when and where they need it. A farmer, for example, relies on the availability of sufficient water at the diversion to his/her farm when s/he needs it. If the water is not available when needed, even if there is an abundance of water in the system as a whole, it will not be of benefit.

While it is important to insure that a river system has sufficient water to handle all of its demands, it is just as important to insure that the river system does not have too much water. The most obvious problem associated with too much water in a river system is flooding, which can endanger both the environment and the lives and property of those who live within or downstream of the river basin. Insuring that a flood does not occur can be difficult since the amount of water that flows into the river system cannot be controlled. A large rainstorm upstream or a warm day that causes the upstream snow

pack to melt can cause a great deal of water to be introduced into the system in a short period of time. While it would be possible to greatly decrease the potential of a flood by decreasing the amount of water in the river system, this would adversely effect other considerations such as the amount of water available for consumption and power generation. In addition, sufficient water must be stored in the system to offset future drought conditions. Thus, a balance must be maintained wherein sufficient water is stored in the system to meet most demands while reducing the risk of flooding.

Another area in which water quantity is used within water resources deserves special mention not so much because of its frequency of use but because it shows the diverse ways in which water quantity can be used. In order to control mosquito populations in and around its reservoirs, the Tennessee Valley Authority fluctuates the level of water in its reservoirs by a few feet at regular time intervals. This causes any mosquito larvae that are in the shallow water near the shore to become stranded on land and die when the water levels are lowered and drowns new mosquito eggs that have been laid on shore when the water levels are again raised. While it will not result in the complete elimination of the mosquito population, it significantly lowers it.

2.2.3. Water Quality

Water quality is concerned with the composition of the water itself and how this matches the needs of its users. Users of water include both humans and the plants and animals that reside in and around the water. There are many factors that determine the quality of water. The most obvious is the amount of man-made pollutants in the water. These pollutants might originate from nearby farms, factories or individuals. In addition to pollutants, the amount of salt in the water is also a measure of the water's quality.

Water quality can also measured based on more subtle characteristics of the water. Temperature and dissolved oxygen are two such examples. Proper temperature and levels of dissolved oxygen are necessary to insure the health and propagation of fish within the river system. For example, if the temperature gets too low or the amount of dissolved oxygen too low, fish will die. As it turns out,

this is exactly what can happen when a dam is introduced into a river system. Since the water that flows from a dam may be released from the bottom of the reservoir, it tends to be very cold and lacking in oxygen. In order to mitigate the effect on fish, the water needs to be heated and oxygenated.

2.3. USBR

The main set of users involved in this study were water resources engineers from the United States Bureau of Reclamation (USBR). The USBR is a branch of the United States Department of Interior and is responsible for the management of many of the various water and related resources in the 17 western states (Figure 1). The USBR administers 348 reservoirs with a total storage capacity of 245 million acre-feet. An acre-foot is equal to 325,851 gallons of water and can supply a family of four with enough water to last one year. In addition to the 348 reservoirs, the USBR also operates 58 hydroelectric power plants that generate 42 billion kilowatt-hours annually and deliver 10 trillion gallons of water to more than 31 million people each year. It also manages 308 recreation sites [USBRb, 2002].

Figure 1. USBR States and Regions [USBRa, 2002].

2.3.1. Regions

The 17 states that contain water systems managed by the USBR are grouped into five regions: the Great Plains, Lower Colorado, Mid-Pacific, Pacific Northwest and Upper Colorado (Figure 1). The Upper and Lower Colorado were the only regions considered for this study. The Upper Colorado contains the upstream portion of the Colorado River Basin and the Lower Colorado contains the downstream portion of the Colorado River Basin. The boundary that separates the two is meant to roughly equalize the amount of water stored in each region. There are about 50 reservoirs in both regions, with one large storage/hydropower reservoir in each. The Upper Colorado contains Lake Powell and Glen Canyon Dam, while the Lower Colorado contains Lake Mead and Hoover Dam. Together, these reservoirs hold about 98% of the water in the Upper and Lower Colorado.

The Upper and Lower Colorado Regions are responsible for the management and allocation of the water within their portion of the Colorado River. This means that they must maintain the appropriate quantity and quality of water to insure that the demands placed on them by the federal government and the individual states and users in their regions are met. This task of managing their individual regions

involves optimizing the available resources given the constraints imposed on them. For the Upper and Lower Colorado regions, this requires both inter- and intra-basin considerations. The inter-basin considerations are generally related to the allocation of water between the regions and between the United States and Mexico. The intra-basin considerations are generally related to the allocation of water to individual states and users who are entitled to water in the basins.

2.3.2. Users

There were essentially two groups of users from the USBR with whom I interacted as part of this study. There was a small group of engineers with whom I worked closely while designing, implementing and updating the languages. They used the languages extensively and provided a great deal of input throughout the duration of this study. Their experiences with and programs written in the first language were instrumental in the design of the second language. In addition to these engineers, there was another larger and more disparate group who used and provided feedback on both languages, although they were not involved in their design or prototyping. These engineers were colleagues of the first group of engineers and would use the languages as part of their jobs.

2.3.3. Policies

While there are some general goals that the regions seek to achieve in the management of their river systems, which include power generation, water supply, recreation and conservation, there are also specific policies or laws that they are required to follow. These policies have been in existence for many years and are intended to ensure that the water within the Colorado River is allocated and managed such that entitlements, safety and power generation are assured [Nathanson, 1978]. There are a considerable number of policies that apply to the water in the Colorado River Basin, although there are five main ones that apply to the Colorado River as a whole and that were used as the basis for the initial study. In the second study, some of the policies that apply to the smaller sub-basins were also used. To give a flavor for these policies, a brief discussion of the five main policies follows.

Upper Basin Rule Curve. The Upper Basin Rule Curve policy is concerned with ensuring that the correct amount of storage is maintained in the Upper Colorado Basin reservoirs. These rules set the storages for each Upper Basin reservoir based on the amount of water in Lake Powell and the total amount of storage in all of the Upper Basin reservoirs. Lake Powell is downstream of all of the other reservoirs in the Upper Basin and is also the largest reservoir in the Upper Basin. Therefore, its storage dictates the storages for the other reservoirs. If the storage in Lake Powell is high, the other reservoirs must decrease theirs and if Lake Powell's storage is low, the other reservoirs must increase theirs. This, of course, leads to a potential problem. If Lake Powell has too much water, decreasing the water in the basin's upstream reservoirs will raise Lake Powell's storage level and if Lake Powell has too little water, increasing the water in the basin's upstream reservoirs will decrease the inflow into Lake Powell. Both of these can be addressed by manipulating the outflow from Lake Powell. Yet, doing so will affect the amount of water in Lake Mead, which is addressed in the next policy.

Equalization. Lake Mead and Lake Powell are the main storage reservoirs in the Lower and Upper Basins, respectively, and maintaining equal storages is important to those who rely on the water. The Equalization policy is an attempt to keep the total storages in the two basins essentially equal [Nathanson, 1978]. If this policy did not exist and the Upper Basin was greedy, it could simply decrease or even halt its outflow from Lake Powell. This would result in a decrease in the amount of water into Lake Mead, since its main source is Lake Powell. Of course, the fact that Lake Mead is downstream of Lake Powell means that Lake Mead cannot give water to Lake Powell. Therefore, if Lake Mead has a great deal more water than Lake Powell, Lake Powell will need to decrease its outflow into Lake Mead and/or Lake Mead will need to increase its outflow. The former must be done such that sufficient water is allowed to flow between the two reservoirs in order to insure that recreation, irrigation and fish habitat are not threatened. The latter must be done such that an excessive amount of water is not released from Lake Mead.

Mead Flood Control. The Mead Flood Control policy is concerned with keeping the storage of Lake Mead at a level that will minimize the chance of a flood while maintaining sufficient storage to

meet the demands of hydropower generation, long-term storage and recreation. This entails keeping a minimum of 1.5 million acre-feet of unused storage to act as a buffer in case of large amounts of runoff or outflow from Lake Powell. In addition to this buffer, the amount of storage is allowed to fluctuate based on the time of year. From January to July, the storage must be increased in order to allow for sufficient water to meet demands throughout the summer. During the rest of the year, the storage in Lake Mead needs to be decreased in order to allow for future increases in the flow of water into Lake Mead.

Surplus. The Surplus policy is concerned with the allocation of water when a surplus water condition has been declared. A surplus water condition may be declared in October if it is determined that there is more than enough water in the Colorado River Basin to meet the demands for the coming year. If this surplus exists, the excess water is divided among the main Lower Basin states (California, Nevada and Arizona) for the upcoming year.

Shortage. The Shortage policy is concerned with the allocation of water when a shortage water condition has been declared. A water shortage condition may be declared in January when it is determined that there is not enough water to meet all the demands and entitlements in the Colorado River Basin for the coming year. The amount of the projected shortage determines which Lower Basin states must reduce their allocation and by how much.

2.4. Application

The application for which the languages were built is called *RiverWare*. *RiverWare* is a general-purpose river basin modeling tool that was designed and developed by CADSWES and is currently being used by the USBR and TVA to model their river basins. It employs a graphical user interface for use in constructing and updating river basin models. It also contains simulation and optimization processors for use in the actual modeling of the river basins.

Predecessors of *RiverWare* have existed in one form or another for many years. The application upon which *RiverWare* was originally based is a basin-specific application that hard-coded all physical attributes and policies of the river system. Between this application and *RiverWare*, a number of

applications whose functionality has become progressively more general and powerful have been developed. In order to provide an historical context for this study, a brief description of these systems follows.

2.4.1. CRSS

The Colorado River Simulation System (*CRSS*) application [Schuster, 1987] was created in the early 1980s to allow its users to model the Colorado River Basin in order to schedule, forecast and plan reservoir operations. It is a program in which data are input into the system from files and the program's output is written to files. Since *CRSS* was created to model the Colorado River Basin, many of the characteristics of the basin are hard-coded into the application. These characteristics include the topography of the basin itself, the methods for calculating evaporation, bank storage and other reservoir-specific information, and the policies by which water is allocated.

Originally, *CRSS* adequately modeled the flow of water within the Colorado River Basin. The policies it included also worked well. Yet, as *CRSS* aged and new information was gained about the basin or the policies, *CRSS* had to be updated. This often proved difficult given that the code that modeled the hydrology and the code that modeled the policies were tightly coupled. This led to software that was brittle and prone to incorrect results. In addition, *CRSS* was designed and coded to model the 'Big River,' which included the main stem of the Colorado River and its eleven main reservoirs. It could not be used to model any of the many sub-basins within the Colorado River or any river basins outside of the Colorado River. In order to address these deficiencies, it became obvious that a more general river basin modeling program was needed.

2.4.2. RSS

CADSWES was selected to work with the USBR to design and develop this new program. Work on the project was started in 1988. In keeping with the spirit of its predecessor, the program was named *River Simulation System* (*RSS*) [RSS, 1992]. As it turned out, *RSS* had two incarnations and a troubled development process.

2.4.2.1. LISP

RSS was designed as an object-oriented simulation system fronted by a graphical user interface. It was programmed using LISP. The main screen was a canvas or workspace on which the river basin model was constructed from a palette of objects that represented river basin related entities. These entities included reservoirs, diversions, confluences and reaches. The model itself was constructed by selecting objects from the palette and dragging them onto the workspace. Once on the workspace, the objects could be named and linked together to form a node and arc representation of the topology of the target river basin. The arcs between objects represented the flow of water from one object to another. For example, the outflow of the upstream object would generally connect to the inflow of the downstream object.

Once the river basin model was constructed, the user could "open" the objects and add data and characteristics to them. For a reservoir, data might include its monthly storage values and characteristics might include the manner in which evaporation is calculated and subtracted from the reservoir's storage each month. Once each river basin object had its required information, the flow of water could be modeled using *RSS's* simulator.

RSS's simulator relied on the information stored in each object coupled with a method of solving systems referred to as mass balance [Behrens, 1992, Behrens, 1991]. Mass balance is a technique in which each object is responsible for determining its own state and passing relevant information about itself to its neighbors. A detailed example is included in Section 2.4.3.1. Early progress was promising, although the development of *RSS* was uneven. In addition, the USBR began to feel that the use of LISP would lead to a system that was difficult to maintain and support.

2.4.2.2. C++

Given these developments and given that there appeared to be a viable alternative to LISP, CADSWES and the USBR began to explore the redevelopment of *RSS* using C++. Although there were

clear schedule-related disadvantages associated with starting from scratch, the long-term benefits of using an industry accepted language were believed to outweigh any advantages of staying with LISP.

With the decision to use C++, development of *RSS* continued until a working system was completed. Like the LISP version, the C++ version included a graphical user interface, a palette of river basin objects and the ability to model generic river basin systems [Reitsma, Sautins and Wehrend, 1993]. In addition, it included a language with which policies could be added to a simulation. This language was needed to allow the USBR to model the various policies that were on the Colorado River Basin. It was also the initial implementation of the application-specific language that would be the subject of these studies.

2.4.2.3. *RSS's* Rule Language

The *RSS* rule language was a forward-chaining rule language that was used by the *RSS* rule-based simulator to effect how water was allocated during a simulation. A brief discussion of this language will be presented in order to show its features.

The overall structure of the rule language was relatively simple. Every rule had the form shown in Figure 2. In this and the subsequent figures, the capitalized letters are keywords and the words in angle brackets represent placeholders for user-defined words or expressions. The rule consisted of a header and a body. The header was used to name the rule, define the variable to be computed and define the object or class of objects that were allowed to execute the rule. The rule's name was not used by the simulator. The variable to be computed could be a value on a simulation object, for example "inflow," or a user-defined name. The former was restricted to an object's inflow, outflow, diversion and ending storage. The latter could be any name and provided a means of making a rule into a parameterless function. The object or class of objects that were allowed to execute the rule could be a specific object, such as "Lake Mead" or a class of objects, such as "Reservoir." The body, which was bracketed by a "BEGIN" and "END," was either an unconditional or a conditional assignment statement. See Figures 3 and 4 for examples. The variable in the assignment statement had to match the variable used in the

header. The expression was a numeric expression of arbitrary complexity and the condition, if used, was a boolean expression of arbitrary complexity. Expressions could include literals, calls to rule-specific C functions compiled into *RSS* and values from simulation objects.

```
POLICY <name> TO_DETERMINE <variable> FOR <object>
BEGIN
   <statement>
END
```

Figure 2. *RSS* rule template.

```
<variable> = <expression>
```

Figure 3. *RSS* unconditional rule statement.

```
IF <condition>
THEN
   <variable> = <expression>
```

Figure 4. *RSS* conditional rule statement.

Using this relatively simple syntax, the users were able to express many of their policies. Yet, they found it difficult to express these policies naturally. In addition, once written, the policies were difficult to read and understand. To illustrate this, refer to Figure 5, which contains two rules that were taken from the most recent version of the USBR's working set of rules. Note that these two rules were fairly typical and were the first two rules of the 57 that made up the Mead Flood Control policy. For the full set of rules that made up this policy, refer to Section B.1.

The first rule was used by the object "mead" to set its outflow during January to July. This was expressed fairly clearly in the first part of the rule's condition. The rest of the condition, however, was not as easy to understand. This part of the condition was used to initialize two "cache" variables to zero. Cache variables were a kind of global variable that existed for the duration of the simulation run. They could be set, cleared, read and checked to determine if they existed. The initialization of the cache variables was placed within a condition because the language did not allow multiple statements within a

```
POLICY out1 TO_DETERMINE outflow FOR mead
BEGIN
IF now () <= JULY AND
   clear_cache (“trial_fc_rel”) = 0.0 AND
   clear_cache (“release_level_index”) = 0.0
THEN
   outflow = max (mead.outflow, flood_control_release)
END

{-------------------------------------------------------------------}
POLICY fc_rel1 TO_DETERMINE flood_control_release FOR mead
BEGIN
IF now () <= JULY AND
   trial_fc_rel AND
   trial_fc_rel <= cfs_to_kaf (now (), now_year (),
                               nth (read_cache (“release_level_index”),
                               &meadt.release_level)) AND
   read_cache (“release_level_index”) > 1 AND
   trial_fc_rel > cfs_to_kaf (now (), now_year (),
                          nth (read_cache (“release_level_index”) - 1,
                             &meadt.release_level))
THEN
   flood_control_release = cache (“flood_control_release”,
trial_fc_rel)
END
```

Figure 5. *RSS* Rules.

rule. This restriction was circumvented by making the clear_cache function return zero and comparing this return value to 0.0, which was guaranteed to evaluate to true. Thus, the only part of the condition that was significant from a policy perspective was the first part. If the condition evaluated to true, this rule set mead's outflow to the maximum of mead's current outflow (i.e., "mead.outflow") and "flood_control_release."

"flood_control_release" was calculated by the second rule. Like the previous rule, it could only set a value from January to July. And like the previous rule, it used a string named "trial_fc_rel." While this name was the same as the cache variable initialized in the previous rule, it did not refer to this cache variable. Instead it referred to a rule for the variable "trial_fc_rel" that existed on "mead." The value returned by this function was used in the second part of the condition to make sure it was non-zero. It was then compared to the result of the function "cfs_to_kaf," which was a predefined C function that converted a value from cubic feet per second to kilo-acre feet. This function took three arguments. The first two were dates that were returned by other predefined C functions. The third was a predefined C

function that took a number and an array and returned the “nth” element of the array. The number was derived from the cache variable “release_level_index” and the array was derived from “&meadt.release_level.” “&meadt.release_level” had a similar form to the previously discussed “mead.outflow.” The leading ampersand, however, meant that a list should be returned and used in the C function. The next two parts of the condition were variations on what has been discussed, so a description of them will be omitted.

The assignment statement, however, is worth discussing. Here, the variable upon which the rule was based (i.e., “flood_control_release”) is set to the value returned by the “trail_fc_rel” rule, while at the same time, a cache variable with the same name as the rule variable was created and set to the same value. The setting of the cache variable was used as a means of storing a value associated with a rule without having to re-execute the rule. This was generally needed when the rule computed a new value with each execution.

Note that while not shown, there were three other rules that set “flood_control_release” for “mead.” These rules all had different names and logic and were used to calculate alterative values for “flood_control_release” given different conditions. Multiple applicable rules were not uncommon in this language and were used to express compound conditional expressions. Each applicable rule was executed whenever one applied and were executed in the order in which they appeared in the file. If more than one rule successfully executed, the last rule to execute set the rule’s value.

2.4.3. RiverWare

When it became clear to the USBR that *RSS* needed to be extended, an assessment was made as to whether *RSS* could be used as the basis for this system or if a new application should be designed and implemented. Given the fact that *RSS* had been implemented in a rather hurried fashion and that it served more as a proof of concept than as a polished piece of extensible software, it was determined that a new system should be designed and implemented. This system was originally named the *Power and*

Reservoir System Model (*PRSYM*), although its name was later changed to *RiverWare*. For the duration of this paper, the application will be referred to as *RiverWare*.

Although *RiverWare* is similar in spirit to *RSS*, it is considerably more powerful. Like *RSS*, it is an object-oriented system fronted by a graphical user interface. And like *RSS*, it provides facilities to construct and edit river basin models and run simulations based on user-defined data and object characteristics. It also contains a policy language that works in conjunction with the simulator to allow policies to influence the flow of water within the river basin. Unlike *RSS*, however, it includes many other water resources engineering related features that allow it to more closely model river basins. These include more information on each river basin object, the ability to add user-defined methods to objects, a more powerful simulation engine and an optimizer.

For this study, the simulators, how river basin models are solved and how the policy language is used are of most interest in *RiverWare*. There are three basic modes that can be used to run a model. They are simple simulation, rule-based simulation and optimization. The optimization mode, which employs linear programming to solve the system, does not use the policy language and will not be discussed here. Simple simulation also does not use the policy language, but since it is the basis for rule-based simulation, it will be discussed in order to lay a foundation for the discussion of rule-based simulation.

2.4.3.1. Simple Simulation

Simple simulation is a means of testing different scenarios that might occur on a river basin over some period of time. For example, a user might be interested in seeing how a river basin reacts to a large amount of inflow into the system that could be caused by heavy runoff. Simple simulation uses a method for solving systems called mass balance. Mass balance is a method wherein each river basin object uses information about itself and information from its neighbors to compute its own internal state. Once its internal state is known, it passes any relevant information on to its neighbors, which continue the process. Mass balance is essentially a discrete-event method for solving a set of

simultaneous equations and assumes a system that is fully specified. This means that, for each time step in the simulation, all variables but one are known in advance.

Mass balance is a fairly simple method for solving the river basin models used by *RiverWare*, although to fully understand it, an example is helpful. As a simple example, consider a three reservoir system with reservoirs named A, B and C. Further, say that A is upstream of B, which is in turn upstream of C. This means that A's outflow is connected, and therefore equal, to B's inflow and that B's outflow is connected and equal to C's inflow. Lastly, say that some information about the reservoirs' states is known in advance. A's inflow and storage are known, while its outflow is unknown and only B's and C's storages are known. For this example, we will only solve the system for one time step. This implies that storages from the previous time step (referred to as 'previous storage') have either been provided through user input or were calculated during the previous time step. Note that units are omitted for simplicity in this and the following two examples.

Figure 6a shows the initial state of the system. Here, A's inflow, previous storage and storage are known. This leaves only outflow as an unknown. For B and C, previous storage and storage are known and inflow and outflow are unknown. Because A has only one unknown and B and C have two, the simulator will solve for A's outflow, as is shown in Figure 6b. For clarity, the values that are changed in each figure are shown in italics. Since A's outflow is connected to B's inflow, B is passed an additional piece of information and is now able to solve for its outflow, as is shown in Figure 6c. And, since B's outflow is connected to C's inflow, C is passed an additional piece of information and is now able to solve for its outflow, as is shown in Figure 6d. With this last piece of information, the system is solved for the current time step.

If there were additional time steps that needed to be solved, the reservoir's current storage values would become their previous storage values that would, in turn, be coupled with any user specified data to solve the system for the future time step. This would continue until the system had been solved for all applicable time steps or the system had insufficient information to allow it to

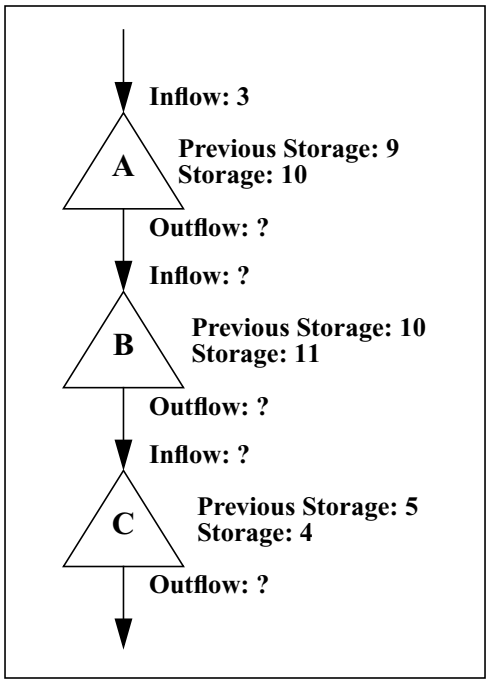

Figure 6a. Initial state of system.

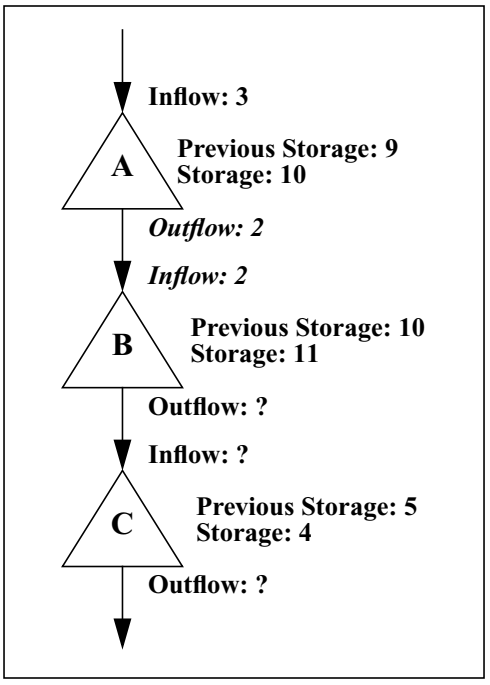

Figure 6b. A's outflow is computed and propagated to B's inflow.

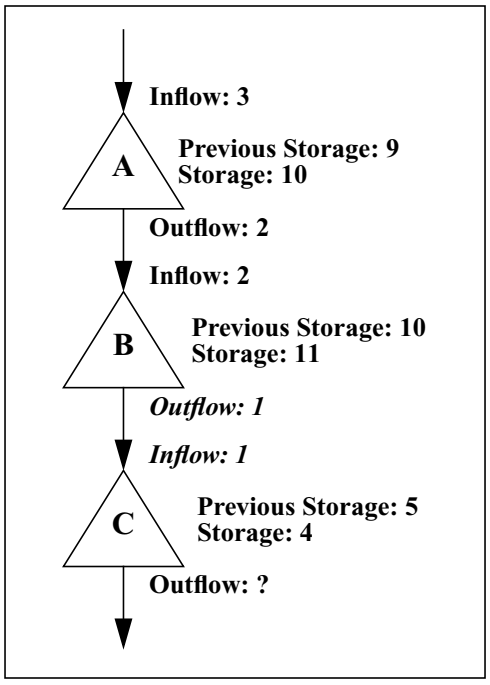

Figure 6c. B's outflow is computed and propagated to A's inflow.

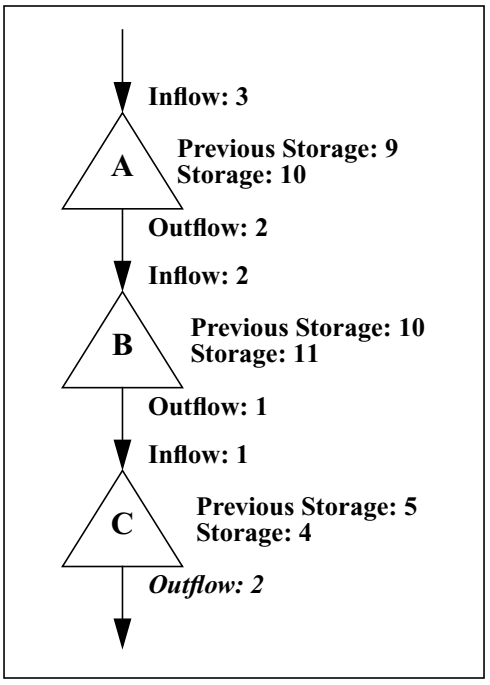

Figure 6d. C's outflow is computed. System is solved.

continue being solved. This latter case would happen in the above example if, for instance, A's inflow, storage or previous storage were unknown.

While this example demonstrates how mass balance can be used to solve a system in which the information progresses downstream, it is also possible for mass balance to be used to solve systems in which information progresses upstream. The direction that the information flows does not, of course, affect the water in the system, which will continue to flow downstream. It is just a reflection of the inputs that are specified in the model. For example, imagine the same three reservoirs as in the previous example. Only this time, A knows its inflow and previous storage, B knows its previous storage and storage and C knows its outflow, previous storage and storage. In this case, C is the only reservoir that has sufficient information to solve itself. Thus, it calculates its inflow and passes it on to B as its outflow. B, in turn, calculates its inflow and passes this on to A as its outflow. A is then able to solve for its storage. At this point the system is solved and the simulation is able to continue to its next time step. See Figures 7a-7d for a representation of the simulation's steps. Note that the water is still flowing downstream, even though the constraints on the system cause the information to flow upstream. By altering the constraints further, it is possible to have information flow both upstream and downstream.

2.4.3.2. Rule-based Simulation

Rule-based simulation is an extension to simple simulation. It also uses the mass balance method to solve the system. However, it differs in that the system is generally not fully specified. Indeed, if it is fully specified, rule-based simulation equates to simple simulation. Rule-based simulation allows the user to extend the information available to a model by introducing a set of policies or rules that influence the way in which water is used within the system. The rules are written using the application-specific languages that are the object of this study. The main benefits of using rules are that the information computed and used by the simulation can contain complex expressions and conditionals and can be based on any information, past, present or future, available in the model.

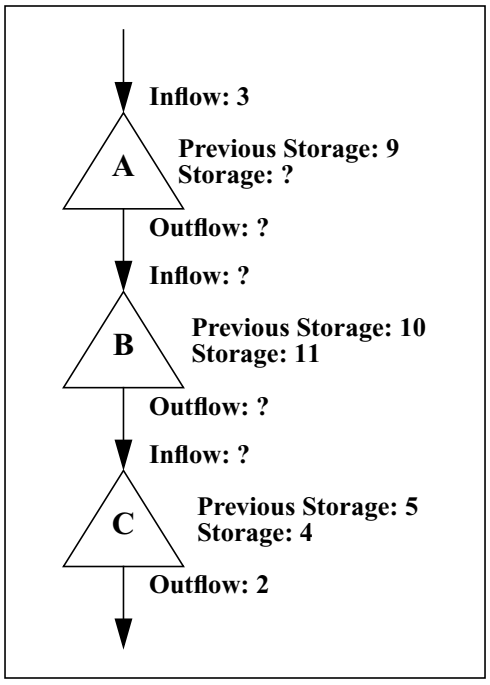

Figure 7a. Initial state of system.

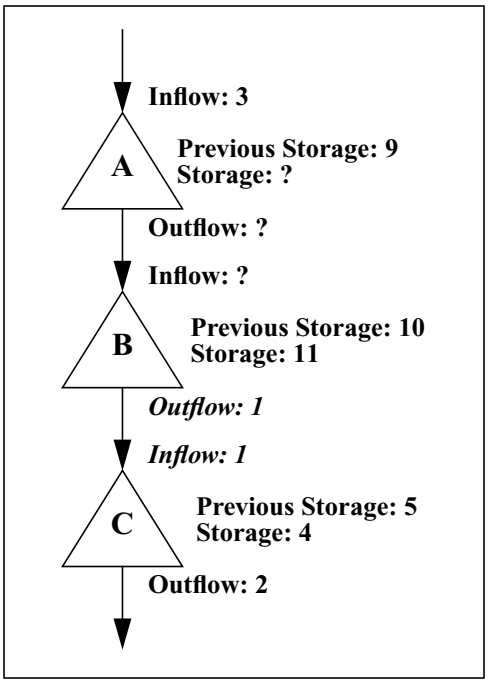

Figure 7b. C's inflow is computed and propagated to B's outflow.

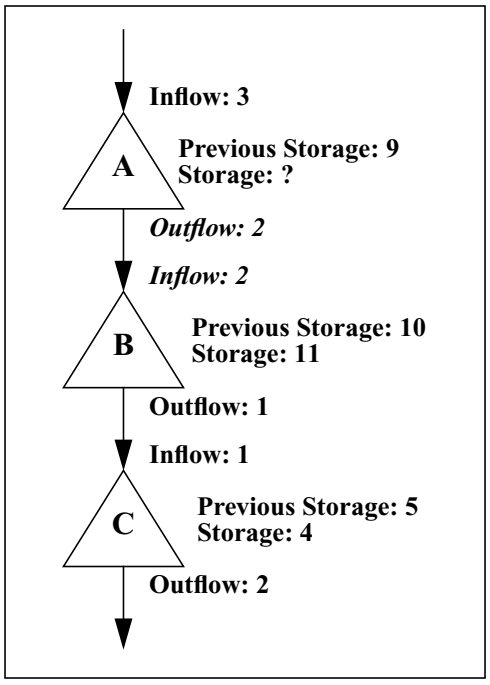

Figure 7c. B's inflow is computed and propagated to A's outflow.

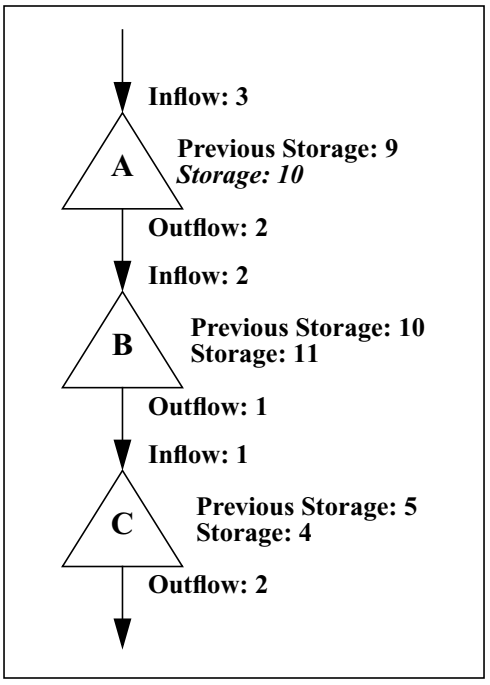

Figure 7d. A's inflow is computed. System is solved.

Figure 8. Rule-based simulation flow of control.

Briefly, rule-based simulation starts with mass balance simulation as described in the previous section. It then adds the ability to include a set of policies or rules that will be used to provide input on an as needed basis. As mentioned above, if the system is completely specified as it must be with simple simulation, rule-based simulation will not need to use the rules to get additional information.

Figure 8 displays a control flow diagram that gives an overview of how simple simulation is augmented by the use of the rules during one time step. The simulation starts by solving for all river basin objects for which it has enough information to do so. At some point, if the rules are to be used, the simulation will be in a state in which it has not solved the entire system and it is unable to continue solving objects using simple simulation. If simple simulation were being used, it would terminate with

an error, as this situation is the equivalent of more equations than variables. With rule-based simulation, however, the simulator is able to look to the rules for additional input.

Rules consist of interpreted code that can be executed on an as needed basis. They can both get information about the state of river basin objects and update the state of river basin objects. In this way, they can use the state of the system to make any changes that correspond to the policies that they are intended to implement. An important point regarding rules, however, is that while they can change the state of the system, they do not necessarily do so. This might happen for a number of reasons, including having insufficient information to successfully execute or having a condition upon which they rely evaluate to false. Once a rule has executed, the simulator again checks to see if there is sufficient information to continue solving. If there is, it does so. If not, it checks for another rule to execute and, if one is found, executes it. This process continues until either the simulator is able to solve the entire system or all applicable rules have been executed and the system has not been solved. In the former case, the simulator can proceed to the next time step. In the latter case, the simulator will stop and alert the user to the fact that it could not solve the system for the given time step. At this point, the user will need to change the existing rules or provide additional inputs in the form of either predefined data or new rules.

A simple example of a small river system will prove helpful in understanding how the simulator and rules interact. In this case, the river system model will include a set of rules that will be used to determine how water should be allocated throughout the system. Figures 9a-9i contain the set of rules and progression of the river system from unsolved to solved. These figures will be used to explain how *RiverWare*'s rule-based simulation works. Along the way, a number of concepts, such as rule priority, will be explained since they are integral to the interaction between the simulator and rules. In addition, some implementation details will be omitted in order to focus on the high level mechanisms underlying rule-based simulation.

Figure 9a contains a set of three rules which will be used for this example. They are ordered by their relative priority, from highest to lowest. Thus, Rule #1 has the highest priority and Rule #3 has the

lowest priority. Priority is used to determine a number of things. It determines the execution order of the rules in that the rules with the highest priority are executed first. If the execution of a rule is not successful, the rule with the next highest priority is executed next and so forth. Thus, rule priorities serve as a kind of tie breaker in cases where more than one rule applies. Priority is also used to determine if a rule can overwrite data that was written by another rule. For example, if a rule wants to set a reservoir's outflow, it can only do so if its priority is higher (i.e., a lower number) than the rule that had previously set that reservoir's outflow. Priorities also apply to predefined data set by the user. These data are entered prior to the start of a model run and cannot be overwritten. Accordingly, their priority is higher than all of the rules. The last place where priorities are used is during value propagation and mass balance. As with simple simulation, values are propagated from one object to another and computed as part of mass balance. When a value is propagated, the new value is assigned the priority of the originating value. When a value is set as a result of mass balance, it is assigned the priority of the value that triggered the mass balance. This is generally the last value to be set on the object. For example, if an object has an inflow with priority 2 and a previous storage with priority 0 (i.e., carried over from the last time step and therefore unchangeable) and a rule with priority 3 sets its storage, mass balance will set its outflow and give it a priority of 3.

Getting back to the rules in Figure 9a, their logic is quite simple and needs little discussion. Briefly, however, Rule #1 will attempt to set B's outflow if it is less than 50. If it is 50 or greater or if B's outflow does not have a value, this rule will not attempt to set B's outflow. Rules #2 and Rule #3 simply attempt to set B's storage and A's outflow, respectively. Note that any attempt to set a value will be rejected if the value was previously set by predefined user data or by a rule with a higher priority.

Figure 6b contains the initial state of the system. Here, only A's inflow and previous storage and B's previous storage are known in advance. The underlined number that is written after each value is the value's priority. In this case, the values are given a priority of zero, which is the highest, because they were set by predefined user data. If this system were being solved using simple simulation, the simulation would halt with an error since there is not sufficient information to continue. Using rule-

based simulation, however, the rules can be used to provide additional information to solve the system. In this case, since the simulator cannot continue solving the system, it must defer to the rules to see if they can provide more information.

The first rule, Rule #1, attempts to execute, but cannot. It needs to know B's outflow in order to evaluate the antecedent of its conditional. Since B's outflow has not been set, it cannot successfully execute. Next, Rule #2 is attempted. Since this rule has no conditions and since B's storage has not been set by predefined user data or by another rule with higher priority, Rule #2 is able to set B's storage to 325 with a priority of 2 as is shown in Figure 9c. While this does indeed provide additional information, there is still not enough information for the simulator to continue solving the system. Therefore, it next tries to execute Rule #3, which like Rule #2 is successful. Thus, A's outflow is set to 50 with priority of 3 as is shown in Figure 9d. As in simple simulation, the value is propagated to B's inflow. Unlike simple simulation, the value's priority is also propagated to B's inflow.

At this point, the simulator has sufficient information to continue solving the system and calculates A's storage, as shown in Figure 9e, and B's outflow, as shown in Figure 9f. Note that when A's storage is set, its value is assigned a priority equal to the priority of the rule that set its outflow (i.e., 3) and B's outflow is set to the priority of the rule that set its storage (i.e, 2).

At this point, the system appears to be solved. This is not the case, however. The system is not considered solved until all applicable rules have been executed. Thus, the simulator needs to defer to the rules again. It again tries to execute Rule #1, which could not execute previously. Since additional information is now known about B, Rule #1 is able to execute. As Figure 9g shows, since B's outflow is less than 50, it is reset by Rule #1 and given a priority of 1.

This additional piece of information now causes the system to have more equations than unknowns and, therefore, become overdetermined. The simulator, in order to balance the state of B, needs to determine which value to change. This decision is made using the priorities of B's values. Previous storage was either provided as user input or was carried over from the previous time step and

cannot be reset. Outflow was set by a rule with priority 1, storage was set by a rule with priority 2 and inflow was set by a rule with priority 3. Therefore, B's inflow will be reset using mass balance. And, since B's outflow, which has a priority of 1, was the last value set on B and the value that caused the re-mass balance, B's inflow will be given a priority of 1. Both B's inflow value and priority are then propagated to A's outflow, as is shown in Figure 9h.

The propagation of B's inflow to A's outflow now causes A to become overdetermined. As with B, the value with the lowest priority will be reset. Since outflow was set as a result of a rule with priority 1 and inflow was set by predefined user data, storage, with a priority of 3, will be reset. Figure 9i shows this. At this point, the simulation has completely balanced the system for this time step and can either continue to future time steps or stop.

2.5. Motivation for New Language

Now that the background for this study has been provided, it is appropriate to conclude the chapter with the motivation for the languages that were used for rule-based simulation. While the policy language developed for *RSS* had been serviceable, the users were generally not happy with it. They had difficulty expressing their policies using it and only managed to translate one of their main *CRSS* policies into it. This was, in part, due to the fact that they had trouble extracting the logic of the policies from *CRSS*. It was also due to the fact that once extracted, they found it difficult to translate these policies into the language. While there were a number of reasons for this, some of the more significant ones were the language's awkward syntax, its reliance on recursion, its inclusion of basin-specific features and a lack of any significant programming support. In addition to having trouble expressing their policies in the language, the users also found it difficult to understand these policies once they were written. Thus, when the decision was made to replace *RSS*, the users also decided to replace the rule language.

```
Rule #1
   if (B's outflow < 50) then
      B's outflow = 50
   endif

Rule #2
   B's storage = 325

Rule #3
   A's outflow = 50
```

Figure 9a. Example rules.

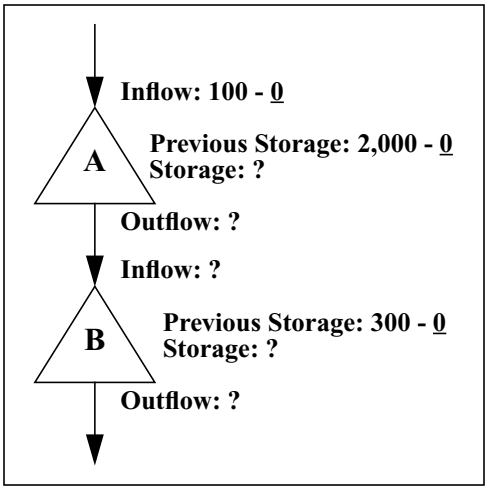

Figure 9b. Initial state of system.

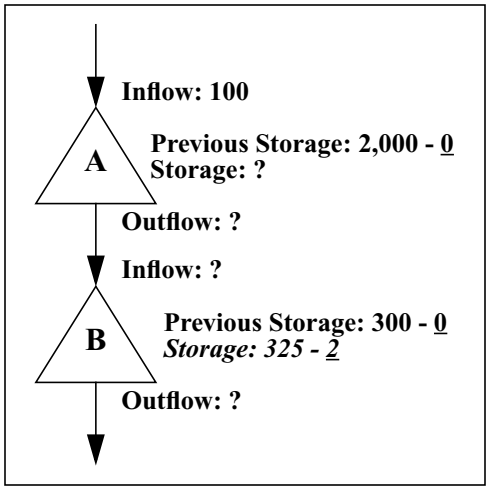

Figure 9c. Rule #2 applied.

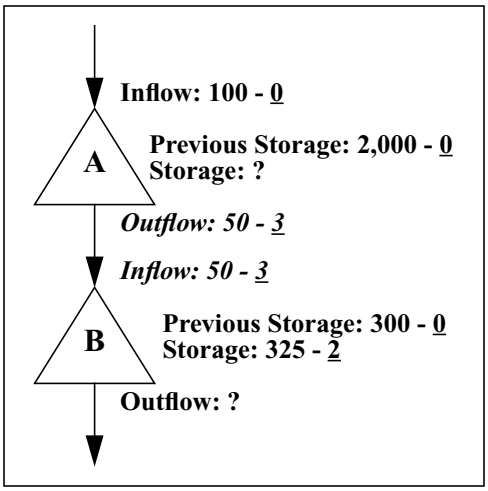

Figure 9d. Rule #3 applied.

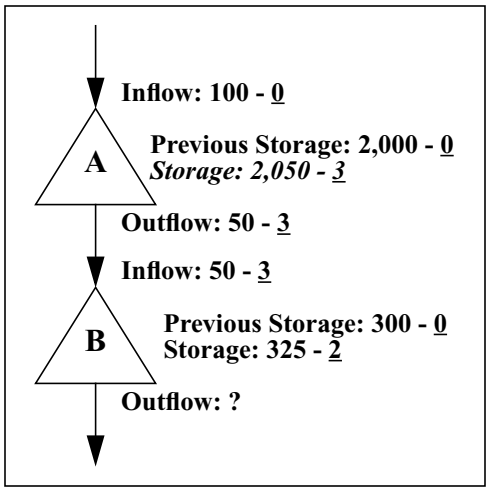

Figure 9e. A's storage is computed.

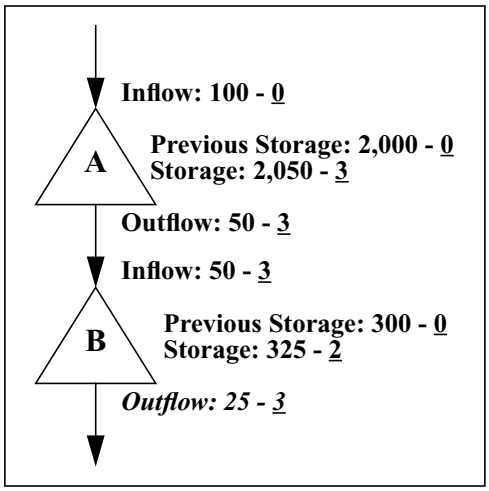

Figure 9f. B's outflow is computed.

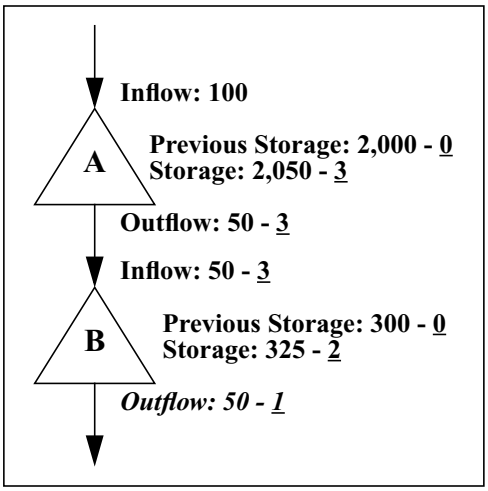

Figure 9g. B's outflow is overridden by Rule #1.

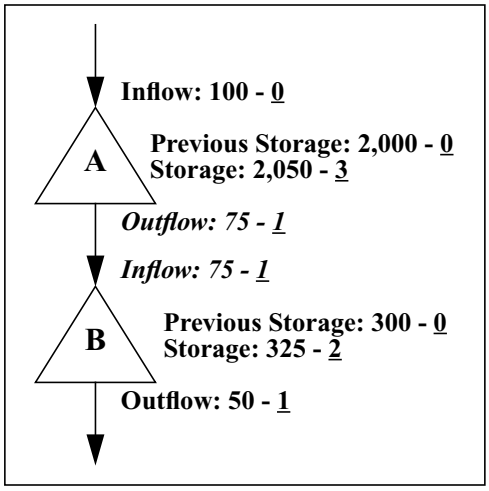

Figure 9h. B's inflow is recomputed and propagated to A's outflow.

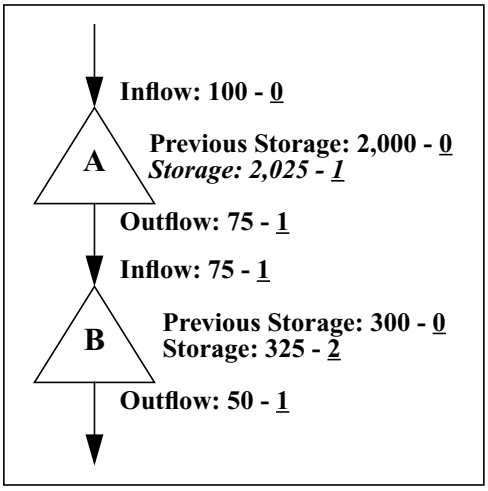

Figure 9i. A's storage is recomputed. System is solved.

Chapter 3

Case Study 1

3.1. Introduction

The process used to design the first *RiverWare* policy language [Wehrend and Reitsma, 1995] was largely based on problem-centered design [Lewis, Rieman and Bell, 1991]. This methodology puts user goals, generally referred to as target problems, at the center of the design. The problems are then used to drive the design and select between design alternatives during the design of a language and accompanying programming environment. While the methodology worked well, there were times when I had to improvise. Accordingly, the design process contained elements that were both experimental and iterative.

The main reason the design process did not proceed as smoothly as anticipated was that the target users had only a high level understanding of the policies that they needed to express using the language. Although they had used the policies for many years in both their use of *CRSS* and in their management of the Colorado River, they were unclear about many of the details of the individual policies and were often unsure how they should interact when combined. As will be discussed later, there were good reasons for this and this new policy language would assist them in resolving these issues.

Despite the uncertainties associated with the design of the language, the process yielded a number of positive results. The resulting language was successfully designed, developed and used to express the entire set of policies that were embedded in *CRSS*. As a result, the USBR was able to use *RiverWare* to duplicate the results of *CRSS* and replace *CRSS* with *RiverWare*. In addition, a number of valuable lessons regarding the design of application-specific languages were learned.

The following sections will discuss the first case study in detail. They will include information on the participants, their problems, the design context and the design process. Lastly, they will present a high-level description of the resulting language, its accompanying programming environment and language processor.

3.2. Participants

3.2.1. Initial Users

The initial set of users were water resource engineers from the USBR whose primary function was to manage the flow of water through the Colorado River. These engineers included representatives from both the Upper and Lower Colorado River Basins. They were familiar with the policies that needed to be expressed using the new policy language. This familiarity came both from their day to day management of the Colorado River and from their use of *CRSS*. They primarily used *CRSS* to run simulation models for the Colorado River, although they occasionally had to make changes to its source code as bugs and policy problems were found. In addition to their use of *CRSS*, they had been involved with the design of both *RSS* and *RiverWare*.

These engineers had varying degrees of programming experience. Some had a moderate amount of procedural programming experience using FORTRAN. Others had almost no programming experience. In addition, none of them worked as programmers as their primary vocation. In general, they would program when needed as a ancillary part of their job. As a result, their time spent programming tended to be sporadic.

An important point regarding these users is that they were very interested in the design and development of the policy language. This was in part because once the language was fully developed, they would be using it as part of their job. It was also because they wished to discontinue using *CRSS* to model the Colorado River and the only way to do this was to use *RiverWare* and its new policy language to show that *CRSS* was no longer needed. This motivation helped to mitigate some of the problems associated with the lack of clarity regarding the policies to be expressed.

3.2.2. End Users

Another set of users appeared after the language had been designed and developed. These users were from the Colorado River Modeling User Involvement Group (CRMUIG). The CRMUIG includes representatives from the seven states that are entitled to Colorado River water. They are the USBR's customers and eventual users of *RiverWare*. Their job is to see that the states that they represent get the water to which they are entitled. Like the USBR, they want to use the language to experiment with policies. Unlike the USBR, however, they are serving the interests of their states and are likely to try to interpret the policies to their state's benefit.

A notable characteristic of this group was that they were involved in neither the design nor the development of the language. They did not use the language until after it was substantially finished. In addition, they generally had less programming experience than the USBR users. These characteristics were helpful in a number of ways. First, they were able to use the system and give an assessment that was free from any potential bias associated with the heavy use and ownership of the USBR users. Also, since they had less experience programming and with using the environment in which *RiverWare* was run, their assessment helped to give an accurate picture of how the ultimate target users would view the language and accompanying environment.

3.3. Target Problems

The users had a number of goals for the language system. Some of these goals related to specific policies that they needed to represent using the language and others related to the manner in which they wanted to be able to use the system. These goals were used to focus the design of the language system in order to maximize its ability to meet their needs.

3.3.1. Programming Language

The target problems that the language would have to support were the five policies discussed in the previous chapter. These policies were pieces of legislation that were written into law many years ago and were expressed in a form suitable for lawyers. They were not expressed in such a way that they

could be used to manage the Colorado River without having to make judgment calls regarding issues that were not specifically addressed in the law. For example, the Reservoir Equalization policy states that Lake Powell and Lake Mead must maintain approximately equal storages and that if Lake Powell has too much water, it must release some into Lake Mead in order to restore the storage equilibrium. The policy does not, however, state what should be done when Lake Mead has more water in storage than Lake Powell. This latter point needed to be taken into account both when managing the physical river system and when running simulations on that river system.

In addition to the law, there was also an interpretation of these policies that was embedded within *CRSS*. These interpretations were written in FORTRAN and were deeply intertwined within the basin-specific simulation code. This was coupled with the fact that the code was acknowledged to be buggy and the policy representations imperfect. Despite all of this, these were the interpretations that needed to be used to demonstrate that *RiverWare* produced the same results as *CRSS* given the same inputs.

As a result of this, the process of determining the definitions of the problems was rather slow. This was in part because it was difficult to extract the policies from *CRSS* given the many interdependencies within the code. This was also in part because as the policies were extracted, bugs and logical errors were found in the *CRSS* code and the USBR users felt that they needed to be corrected in order to meaningfully compare *CRSS's* results with those produced by *RiverWare*.

A final interpretation of these policies was the USBR users' conceptions of them and how they should be used to effectively and fairly manage the Colorado River. This final interpretation, while important and to be used in the future, could not be used until *RiverWare* could duplicate *CRSS's* results. Therefore, this view of the policies, while discussed on many occasions, did not play a significant role in the design of the first policy language, even though this was the version of the policies that the users ultimately wanted to express.

3.3.2. Programming Environment

In addition to the ambiguity surrounding the policies to be represented in the language, there was also uncertainty regarding what type of programming environment would be needed. The users had some ideas about what they wanted in terms of some specific tools, but they really did not have a coherent idea of what was needed. To a large degree, the programming environment was an afterthought for both the users and me.

The target problems discussed in the previous section were of little help, since they pertained to the users' policies and therefore applied to the language. I was, however, able to gather information about programming environment needs through discussions I had with the users. Most of these discussions centered around what the users wanted to do with the policies as they were created and modified.

One of the main pieces of functionality that the users wanted was the ability to make changes to the policy descriptions between simulation runs without making changes to the *RiverWare* executable. This was necessary to allow them to create, test and experiment with different versions of their policies and required a language that could be interpreted by *RiverWare* at run time. They also needed the ability to change the priorities of the rules between simulation runs. This was similar to changing the policies themselves, but allowed them to see the effects of re-prioritizing policies. The users also made requests for specific functionality that they felt would help with the creation and maintenance of their policies. Examples of these included version control and a means of comparing different versions of the same policy.

3.3.3. Language Processor

The language processor is that part of the language execution environment that is responsible for loading the policy statements into *RiverWare* and for executing these policies when needed. Loading the policy statements into *RiverWare* is a reasonably straightforward endeavor and can be accomplished without much interaction with *RiverWare*. Executing policies, on the other hand, is much more

involved. As was discussed in the previous chapter, the addition of policies necessitated a fundamental change to the simulation process. Rule-based simulation, which has two fundamental parts, needed to be added to *RiverWare*. The first part is the main rule-based simulation engine. This engine is an extension to simple simulation with the added ability to request the execution of policies when needed. The second is the logic responsible for executing policy statements as dictated by the policies' priorities.

In addition to these two parts, rule-based simulation also required a large number of changes to the underlying objects in *RiverWare*. Prior to rule-based simulation, the river basin objects did not have the concept of priorities. Their individual variables, such as inflow, outflow and storage, were simply numbers with associated units. With the advent of rule-based simulation, these variables needed to have priorities associated with them so that situations in which an object was overdetermined could be consistently handled.

Finally, rule-based simulation caused a change in the way concepts such as overdetermination and underdetermination were defined. With simple simulation, an overdetermined or underdetermined object would cause the simulation to generate an error. With rule-based simulation, an underdetermined object would cause the simulator to request the execution of one or more rules in order to remove the underdetermination. An overdetermined object would require the simulator to re-mass balance the object based on its variables' priorities.

3.4. Design Context

3.4.1. *RiverWare*

RiverWare was the application within which the policy language would be included. Although many parts of *RiverWare* were completed at the time I began work on the language, it was still being developed and most of the features that would be necessary to integrate a policy language into the application were absent. This included the rule-based simulation engine and routines to access the state of *RiverWare* objects during simulations. In addition, since *RiverWare* was under development, even the

portions not directly related to a future policy language were not completely stable and tested. On the positive side, however, there was an established development environment that allowed developers to work independently. In addition, the fact that *RiverWare* was being developed meant that there was the support and expertise necessary to insure that the new rule language could be successfully integrated into *RiverWare*.

3.4.2. Real-world Constraints

This case study and the subsequent case study, discussed in Chapter 5, were made both more interesting and more difficult due to the fact that they took place in the context of a real-world programming project. This meant that the language needed to be designed and developed for a group of real users who needed to use *RiverWare* and this language to do their jobs. It was not enough to develop an experimental language and learn some interesting lessons. The language had to be acceptable to the users both in terms of its syntax and in terms of its ability to represent their policies. It also had to be efficient enough to allow them to run their simulations quickly. This last requirement is admittedly vague and while there was no predefined time limit, the users would need to model complex systems over 80-year simulation time horizons in at most hours rather than days.

In addition, the language needed to be designed and developed for use within a commercial application under genuine time and budgetary constraints. This meant that decisions would have to be made that were not always motivated by what was considered in the best interests of language design. This does not mean that all of the decisions made were poor. Some of the quick decisions turned out to be good ones, but even those that were not so good were generally the best decision for the given context. In either case, since this type of language design process is likely to be similar to other real world language design processes, the lessons learned should be of interest to others.

Another aspect of the problem that made it both interesting and difficult was the fact that the target users were not professional programmers who nonetheless needed a language that they could use to program some reasonably sophisticated numerical and logical problems. The language had to be

designed and developed to such a degree that it would allow them to express their policies and make changes to them as needed. A purely experimental language or one that could only be used by trained programmers would not suffice.

Another relevant consideration for these case studies was the fact that I was reasonably new to the domain of water resources modeling and very new to the policies that the language needed to be able to represent. In addition, although I had done a great deal of development work on *RSS*, I had not been involved in either the design or programming of *RiverWare*. Lastly, I had never designed a programming language before and had few resources on which to rely within my organization. Given all of this, the process by which the final language and its accompanying environment was arrived at was at times uneven and included both good and bad decisions.

3.5. Design Process

As stated at the beginning of this chapter, the entire design process was not defined in advance. I had decided that I would use problem-centered design to the extent that I could [Lewis, Rieman and Bell, 1991], but I was not sure how well this approach would work in the context of a real language design. And since I was using the case study to both design a language and try to gain insight into the design process, I felt free to experiment with the process as seemed appropriate.

In addition to a programming language, a programming environment and a language processor had to be designed, developed and integrated into an existing, although unfinished, application. As a result, it is difficult to discuss the design process in a neat, linear fashion. Generally, investigations into alternative languages, programming environments and language processors overlapped and decisions regarding these areas were made when enough information was available. This was in part because I was dealing with real users who could not always provide prompt feedback on important design-related issues and in part because I was working under often ambitious time constraints. It was also because I was learning a great deal about the domains of water resources engineering and application-specific language design.

Given the manner in which the design progressed, I will start by discussing the beginning of the process and how it motivated further investigation into the language, its programming environment and the language processor. I will then discuss the design decisions and processes that related to the sources of information that I used to aid in my design. These areas included the USBR users' problem descriptions, *RSS's* rule system, expert systems and rule languages. I will conclude with a discussion of the design decisions made, the motivations for these decisions and how the system was developed and refined.

3.5.1. Getting Started

The design process started slowly. I first needed to gain an understanding of the users and their problems. I started this process by getting together with two of the main USBR users to talk about what they felt they needed in a language and an accompanying programming environment. At this stage, I did not focus on the language processor since I was not yet concerned with this part of the language and, even if I had been, it is unlikely that the users would have had any useful input at this point in the design process.

There were a number of areas I wanted to explore at this phase of the design process. These included overall perceived needs, opinions about the approaches used by *CRSS* and *RSS*, and the types of policies that would need to be implemented using the new language. I started with the quite general area of perceived needs as a way to allow the users to tell me what they needed without trying to tie it too closely to what I understood they needed. I did not wish to bias them with my preconceptions.

I also wanted to find out how they felt about the two systems that they had used in the past to express policies for their Colorado River Basin model simulations. As discussed in Chapter 2, *CRSS* was a system in which all basin properties and policies were intertwined and hard-coded. Given that they wished to supplant this system with *RiverWare*, most of the feedback regarding *CRSS* was concerned with the ways in which they wanted the new language to be different. The main features that they wanted for the new language were the ability to change the policies without requiring the host

application to be rebuilt, the ability to completely separate any basin-specific properties from the simulator and the ability to individually prioritize their policy descriptions.

The users also had some specific feedback on the second system, the *RSS* rule language discussed in Section 2.4.2.3. While they had been able to express some of their policies using it and use these in *RSS* simulations, they felt that the language was difficult to use and required them to represent many of their policy-related concepts unnaturally. For instance, the language did not contain functions. Functions had to be represented as rules. While this worked, it made the language difficult to use and blurred the line between what were true high-level policies and what were simply helper functions. Also, no procedural looping constructs were provided. Looping had to be accomplished using recursion. While the users could create their rules using recursion, they found it confusing. Lastly, the users wished to have local variables. The *RSS* rule language had a kind of local variable, although it was difficult to use, made their rules more difficult to read and, if used improperly, could cause unexpected results.

The users also had an opinion about how *RSS's* rule language was integrated into the *RSS* graphical user interface. *RSS's* rule language was primarily textual. The only links to the GUI were a menu option to load a rule file and a window that displayed the status of the load. The users wished to have a programming environment that provided more support and included, at a minimum, the ability to load, view, edit and re-prioritize their policies.

In addition to how the policies were represented in *CRSS* and *RSS*, I also tried to learn about some of the specific policies that would be represented using the new language. At this stage, I knew nothing about the policies and was trying to get a good overview of what they were and how they would interact. As a result, their descriptions stayed at a high-level and were primarily verbal.

3.5.2. Problem Descriptions

3.5.2.1. Programming Language

The language itself was of obvious interest to both the users and me. It would need to be expressive enough to represent their policies. Yet, given that the users were not trained programmers, it would also have to be reasonably easy to read, write and maintain. As stated above, the users had some ideas about what they wanted the language to be like. These included both what they wanted in the language and what they did not want in the language. I had no particular preconceptions about what the language should look like. My main goal was a language that would be as usable as possible. Options included a purely textual language, a purely visual language and some combination of the two. A visual language or one that incorporated visual elements held some appeal in that I thought it might be able to provide the users ways to more easily represent and understand their programs [Green and Petre, 1996, Repenning and Sumner, 1995, Green, 1995, Green and Navarro, 1995, Selker, 1988, Selker and Appel, 1991, Böcker, Fischer and Nieper, 1986]. The users were open to whatever option allowed them to express their policies most naturally.

As stated in the previous section, I had talked to the users and had received verbal descriptions of some of their policies and how they might interact. Unfortunately, these verbal descriptions were not enough to allow me to design a language. It was clear that in order to really understand the underlying policies, I had to have more complete, written versions. The users were happy to provide these.

The written descriptions of the policies were helpful in more fully specifying how the users felt the policies should actually be expressed. They came in various forms and with varying degrees of correctness and completeness. The forms ranged from prose to equations and generally contained some of each. They also included tables that contained data that the policies used. Most of the policy descriptions were specified at a high-level and did not lend themselves to translation into a programming language. In fact, the descriptions were often sufficiently ambiguous that it was sometimes unclear what was intended even from a high-level. See Appendix A for examples of these policy descriptions. They contained ambiguous logic, words that could designate numbers or functions

and outright mistakes. This was largely due to the fact that the authoritative source of these policy descriptions was old, buggy FORTRAN.

It should be stated that these descriptions were not ambiguous because the target users did not understand the underlying policies at a conceptual level. Rather, the users were unclear how their high-level understanding of the policies should be turned into unambiguous representations. In addition, as was discovered later, the users needed to write the rules in the context of the simulation to know for sure whether they actually expressed what they needed to. This was in part because they needed to duplicate *CRSS's* results and in part because they often seemed more comfortable with concrete numbers than with abstract concepts.

At first I tried to disambiguate their policy descriptions by using the form in which they were provided. If I was given a policy description in mostly prose, I would use this form as a basis for either a verbal or written exchange of ideas. Generally, I would tell the policy writer where I thought the description was ambiguous or incomplete and ask for clarification. Initially, this seemed to be a reasonable approach given that the users were likely to be able to give feedback on the descriptions in the form in which they were provided. Initially, it also worked in that I was able to get clearer and more complete policy descriptions. Yet it soon became apparent that it would not allow me to define the policies. The main reason for this was that most of the policies contained significant amounts of prose, which exacerbated ambiguity. This, in turn, made it difficult to come to a common understanding of what they meant. The users knew what they thought their policies meant, but this was either not what they actually meant or it was not what I understood them to mean.

As a result, it became clear that a more formal language was needed to unambiguously state the policies. A formal syntax would allow the target users and me to discuss the policies with a common understanding of what a given representation meant. It would also remove a great deal of ambiguity from the policy descriptions and allow us to see what the descriptions really meant.

This language needed to be both familiar and acceptable to the users. The users were familiar with a number of well known languages, including FORTRAN [Leigh and Huffman, 1987] and LISP [Allen, 1978]. Most had experience with FORTRAN from their work with *CRSS* and found it to be an acceptable language. Their opinion of LISP, however, was not so favorable, as was discussed in Section 2.4.2.1. As a result, the users did not want to use LISP or any language that even remotely resembled LISP.

Given this information, I decided to formalize their policies into a FORTRAN-like pseudo-code. I decided against using pure FORTRAN for several reasons. One was because this seemed like an undue burden on the users. Another, and more important consideration, was the fact that the policy descriptions contained concepts that could not be easily translated into FORTRAN. For instance, all of the policy descriptions incorporated numbers with associated units. These units generally related to the volumes or flows of water and the policy descriptions often contained expressions that mixed units. While pure FORTRAN could have been used to approximate numbers with units, it would be an unnatural approximation that would detract from the goal of defining the policies. Another reason for not using pure FORTRAN was that I did not have any intention of making the policy language exactly like FORTRAN and wanted to avoid expectations to that effect.

I used the FORTRAN-like pseudo-code to translate the policy descriptions. I then wrote a short description of this syntax and sent it with the new policy descriptions to the users along with a request that they review and make changes and corrections to them, as necessary. This worked much better than using their original notation and helped me converge on more complete versions of their policies. This convergence, however, was iterative. It took some time to come to versions of the policy descriptions that were both fairly unambiguous and complete. For examples of the policy descriptions that were written in the pseudo-code, see Appendix C. Note that these policy descriptions are closer to their final versions than their initial versions and reflect a great deal of iteration between the users and me.

This process of iterating toward substantially complete and correct versions of their policy descriptions took a number of months. This had a number of benefits. First, it allowed me to gain a great

deal of understanding about the users' target problems, which are quite complex. It also allowed the users to more fully specify and, as a result, understand their policies. As a result, this was a valuable learning experience for the users and me. While it would undoubtedly be preferable to have a complete set of unambiguous problem descriptions that could be used to design a language that met their needs, users may not fully understand the problems that a new language must be able to express. This process, while time-consuming, helped us gain the understanding needed to design an acceptable language.

Another tangible benefit was that a number of desirable language features began to emerge. These included the various control structures, such as *if* statements and *while* and *for* loops, and data types, such as integers and strings. None of this was particularly surprising given that they had used a FORTRAN-like pseudo-code and the problems that they needed to express originated in a FORTRAN program. What was probably of more value was seeing the way in which many of the loops were used to consolidate data and could be replaced with either language statements or functions.

Perhaps some of the most interesting language features that emerged from this process were the need to tie units to values and to insure that only combining and comparing compatible values would be permitted. Of course, neither of these features were strictly necessary in order to allow the users to express their policies. Most programming languages do not support unit compatibility checks and still allow users to successfully create programs that process multi-unit data.

The main reason that these features were considered both useful and necessary was the fact that the policy descriptions made extensive use of numbers in arithmetic, logical and relational expressions. For instance, flows are generally represented in either cubic feet per second (*cfs*) or in acre-feet per month (*af/m*). If two flows are both in *cfs*, they can be either combined or compared without any unit conversion. Yet, if one flow is in *cfs* and the other is in *af/m*, they can only be compared or combined if one of the flows is converted to the other's units or if both are converted to another, compatible unit, such as cubic meters per second (m^3/s). If the units are not compatible, the program will generate unpredictable and incorrect results. The problem can be even worse if a program combines

completely unrelated units, such as m^3/s and m. No amount of conversion will cause these two units to be compatible and any attempt to compare or combine them should be prohibited.

Despite the potential to allow programmers to create difficult to identify bugs related to the combination or comparison of values with incompatible units, few programming languages support unit checking or conversion. Programmers are almost always required to make these checks and conversions themselves. Yet, requiring the programmer to manage the units of all the program's values can obscure the underlying intent of the program and take a considerable amount of time.

The need for unit checking was further reinforced by the USBR users' propensity to combine and compare values with incompatible units in their pseudo-code policy descriptions. These mistakes did not abate even after I showed them examples of these problems, asked them to declare the units for each value used in the rules and functions, and asked them to carefully check their policies for this type of error. It seemed likely that unless the language provided unit checking, the rules expressed in the language would include potentially obscure unit-related bugs.

The task of iterating over the users' policy descriptions proved helpful for quite a while. Yet, it soon became apparent that the benefits of this process were waning. The main reason for this was that the users were unable to fully define their policies without seeing how they worked within the context of rule-based simulations. There were two reasons for this. One was because they needed to see the interactions of the policies during the simulation. The other was they needed to have the simulation produce actual numbers based on the execution of their policies.

This need is quite understandable since they had not used a system exactly like this. *RSS* had been similar, but the rules created for that application were in a much different form and were not considered to be either complete or correct. Even if they had used a system exactly like this, it is unlikely that they would have wanted to spend the time to completely design their rules outside of the context of an actual system. First, this process would have been slow. Second, the users were accustomed to dealing with numbers and needed them to usefully compare *RiverWare* and *CRSS*.

In addition to the language as it related to the policies themselves, the users also needed to have some way to organize the policy descriptions they wrote. This need did not come directly from the users' problem descriptions. It came both during the iteration process and from discussions with them regarding the language, in general, and the policies, in particular. Generally, their individual policies were composed of a number of sub-policies and functions. These sub-components were often specific to the main policy and would either be unusable by another policy or, if used, would potentially cause incorrect results.

3.5.2.2. Programming Environment

The programming environment is the part of the language system that allows users to create, update, display and maintain programs written in the language. It can also allow users to test the correctness of their programs, maintain different versions of their programs and provide any number of domain-specific features. Often, a programming environment can be the difference between a language being used and not used. If designed and developed properly it can greatly enhance the users' programming experience and their ability to quickly and correctly create their code. It can even enhance their ability to read their code. Of course, it cannot make a bad language good, although it can make it easier to use. A programming environment can go a long way toward either making or breaking a language. That being said, I initially considered this aspect of the language design of secondary importance and did not spend much time on it.

As with the programming language, I also tried to get problem descriptions from the USBR users. I tried to go about this both by gathering general information about what they would want to do with the language and by asking specific questions that related to how they thought they would need to make use of language.

The users were not able to give me problem descriptions that came close to the concreteness of the programming language problems. They were, however, able to give me a good idea of some of the basic functionality that would be needed to make their job both possible and easier. The former case

concerned the need to load a set of policy descriptions into *RiverWare* so that the rule-based simulation engine could use them. There would need to be some facility to do this, even if it were fairly primitive. The latter case concerned giving the users features such as editors and debuggers. These were not strictly necessary, since they could either be borrowed from the host system (in this case UNIX) or omitted altogether. The omission of these features might make the users' lives more difficult, but they would still technically be able to perform their job.

A number of programming environment features came to light during the discussions with the users. Their origin differed from the programming language features that were discussed in the previous section. In general, these features came to me as specific requests for functionality and they often came to me in the form of specific descriptions of features. In other words, they often came as "hows" not "whats." For example, one programming environment feature requested was a rule comparison utility, which would allow the users to view and analyze differences between two versions of the same rule.

Given that there was not a great deal of emphasis on the design of the programming environment, the discussions of the identified areas will be brief. As will be discussed later, the programming environment was in large part designed on the fly using my background as a user interface designer and programmer. While this can work, it is a risky strategy, since user interface design is not necessarily an intuitive task. Without a good understanding of the users, their domain and the tasks that they need to perform, a user interface can be designed and developed that will make the users task far more difficult than it might otherwise be. A good design accompanied by programming walkthroughs [Bell, et al., 1994] can help to identify and correct problems early in the design of the language.

As mentioned previously, the ability to load a set of policy descriptions, referred to as *rulesets*, was a fundamental feature that was needed. In addition, it was not enough to simply be able to load a ruleset once, as might be accomplished with the inclusion of the ruleset file on the command line at the time *RiverWare* was started. The users needed to be able to load rulesets into *RiverWare* any number of times so that they could make changes to the rules and see how these changes affected the results returned by the simulation. In addition to loading rulesets, the users also needed to have feedback on the

status of their load. For instance, if the ruleset contained rules that were syntactically incorrect and would not execute, the users should be made aware of this as early as possible. Of course, one could write a language loader that did not do this and instead allowed incorrect rules to be executed. This would be at best annoying and would at worst cause obscure bugs, unexpected results or application failures. *RSS's* rule system provided for both loading rulesets and for checking their syntax.

Another desired feature that came to light in a discussion with the core set of USBR users was related to how rules would be grouped and maintained in files. The users wished to have flexibility in the manner in which rules and their functions could be combined and stored. For example, they wanted to place all rules and functions associated with one particular policy in one location. That location could be a file or a number of files within one directory. In this way, they could easily identify and track their policies. They also wanted to be able to store groups of functions that were used by more than one policy in a location accessible to all the policies. These utility functions would be made into a kind of library. The ultimate implementation of the above user needs would depend on many factors and would have to wait until more information was gathered. Even so, we discussed some possible approaches that included the use of multiple UNIX files that would store the rules and functions and that could be combined where necessary.

As mentioned previously, the users wished to have rules that could be re-prioritized. This would enable them to see the effect of different rule orderings on the simulation's results. In order to make this a useful feature, however, the rules would need to be re-prioritized without affecting the *RiverWare* executable. This could be accomplished in a number of ways. One would be to require the user to edit the set of rules and reload the ruleset. The other would be to allow the user to re-order the already loaded rules. The latter would be more convenient for the user.

The USBR users also requested a version control tool to be used with the language. This would allow them to save multiple versions of their rules and functions, which would provide a safety net in case a particular version became corrupted. It would also provide a historical record of the rules and functions and allow the users to return to an older version as necessary.

Lastly, the USBR users wished to have some facility to help them debug their programs. They originally asked for a traditional debugger. During our discussions, however, it became apparent that a full-featured debugger was not necessary and that they simply needed a means to determine the correctness of their rules and where necessary locate and fix the specific parts of the rules where problems existed.

Most of the functionality identified for the programming environment were not identified with user-defined problems in the same way that they were for the programming language. I had originally tried to get them to give me concrete problem descriptions in the form of use scenarios and descriptions of how they thought they would use the language. This did not work in the way I had hoped. It appeared that some sort of catalyst was needed to get this information from the users. This could be an existing programming environment or a person who would drive a discussion of the possible programming environment needs.

Thus, the problem descriptions that were used were derived from conversations with the users in which I tried to get them to tell me what they liked and disliked about *RSS's* programming environment and how they wanted to use a programming environment for the new language. As with the written examples given for the programming language, this process was also iterative. Generally, the users would mention a feature that they might liked either explicitly or in passing. From here, the discussion would have to be directed in order to get them to describe more fully what they needed and why they thought they needed it. Although, these discussions often touched on how they wanted these features to be implemented, it was generally possible to determine the underlying problem and work from there.

3.5.2.3. Language Processor

I did not attempt to get problem descriptions that related to the language processor. I did not feel that this area lent itself to problem descriptions and focused on the language and environment. I felt that this was an internal design issue that was not directly related to the user.

In part, I was right, although there were a number of areas that related to the language processor that directly affected the user. The main one was the need to separate the policies from the simulator. Rule-based simulation had to be generic to any basin and could not contain any policy-specific functionality. Another need was to insure that rules were executed in priority order. Lastly, there was the need to efficiently execute the rules.

As it turned out, all of the language processor-related needs were accounted for using the programming language problem descriptions and during the investigation of *RSS's* rule system. While it is possible that future language design efforts will require language processor specific problem descriptions, I suspect that few, if any, will need to do so. Either way, the programming language problem descriptions will need to be scrutinized in order to identify any language processor-related needs that may be part of them.

3.5.3. *RSS's* Rule System

At about the time I was interviewing the users, I started exploring *RSS's* rule language, programming environment and language processor. I had a number of reasons for doing this. The first was that I wanted to have a clear understanding of the system that the USBR had used so that I could understand the opinions that they expressed about it. I also wanted to gain as much information as I could from the decisions that had been made by the language writer. Lastly, I wanted to understand how the language processor worked. At the time I started working on this language, I had no experience in language design or the tools needed to develop a language. In addition, I did not have a clear understanding of how the rules written in this language were executed from within *RSS* and how the rules exchanged data with *RSS* during their execution.

3.5.3.1. Programming Language

The *RSS* policy language allowed the users to create portions of a number of their policies. While the users did not find it to be a desirable language, it was flexible enough to allow them to express most, if not all, of their policy descriptions. The main flaw in the language was probably that it did not

allow the users to express their policies in a natural manner. The policies could be represented, although their form was so different from how the users would have liked to express them that they found them difficult to read and write. Most of the users' dislike stemmed from the fact that the language required complex syntax in order to write some relatively simple rules. A small part of the users' dislike, however, could also be attributed to their lack of programming training and the fact that the bulk of their understanding of programming came from their use of FORTRAN.

As is probably evident from the discussion of *RSS's* rule language in Section 2.4.2.3, the language required a considerable amount of effort to write and read. In addition to this, there were some specific problems that were either pointed out by the USBR or were noticed during my evaluation of the language. I will discuss them here and point out those issues that the USBR explicitly complained about.

Some of the main areas that the USBR complained about related to the fact that the *RSS* rule language was not a procedural language. There were no true functions, although they could be approximated. In addition, local variables had to be expressed using cache variables, which in turn had to be initialized within conditionals. Lastly, there were no looping structures; recursion was required to accomplish this.

The users also did not like the fact that the rules were prioritized by the order in which they appeared in the rule file. The later rules took precedence over the earlier rules and, therefore, had higher priorities. This was not explicitly stated in the rules themselves, which could make it difficult to understand the rules' underlying policy. The USBR preferred explicit rule priorities.

In addition to these complaints, there were other problems that were not noticed by the USBR. These included the fact that cache variables would retain their values between entire rule executions within a single simulation run. Thus, a value could be carried over from a rule execution in one time period to a rule execution in a subsequent time period. This provided an undocumented global variable

that allowed rules in one time period to pass values to rules in another, which could lead to difficult to identify bugs.

3.5.3.2. Programming Environment

As mentioned earlier, the *RSS* programming environment was very primitive. It provided two features. One was the ability to load a ruleset into *RSS*. The other was to provide feedback on the loading process. The loading process consisted of using a file chooser to select a file. If the file contained a syntactically correct ruleset, it would be loaded and the user informed of the load's success. If the file did not contain a syntactically correct ruleset, the user would be given a dump of the load errors. This could be extensive if the ruleset contained many errors or omitted certain nesting constructs. The users found this programming environment to be inadequate and wanted an environment that would provide them with more support, as was discussed in Section 3.5.2.2.

3.5.3.3. Language Processor

The *RSS* language processor functioned well, although the USBR users wished to remove the basin-specific logic that was embedded within it. This basin-specific logic included the implicit prioritization of certain rule types, the ability to set values on only a subset of an object's variables and the fact that rules were tied to certain objects and object types. While the presence of each of these features had allowed the users to create a subset of the rules needed to model the Colorado River Basin, they were unlikely to allow for the creation of all their rules or for the creation of rules for other river basins.

The implicit prioritization of certain types of rules occurred as a result of the order in which rules for the same object or object type were executed. Recall from Section 2.4.2.3 that there were four high-level rule types that could be used to set variables for model objects. The variables that could be set were inflow, outflow, diversion and ending storage. The implicit prioritization stemmed from the fact that the rules were executed in the order specified above. The rule writer had no control over this order. Thus, inflow had the highest priority, followed by outflow, diversion and ending storage. The practical

consequence of this was that if an object did not have enough information to solve, it would first look at the inflow rule, assuming its own inflow was not known. If the inflow rule executed, the object would not get a chance to execute any other rule. An example of how this could affect a simulation follows.

Consider an object that knows its beginning and ending storages and knows neither its inflow nor its outflow. Assume also that rules exist for both its inflow and its outflow and that these rules will always execute and set their respective variables. If neither the inflow nor the outflow can be set without rule execution, the simulator will look to the rules. In this case, the inflow rule will successfully execute and set the object's inflow. This will give the object enough information to solve itself and compute its outflow using mass balance. But what if the rule writer wanted to have the outflow rule take precedence over the inflow rule? This cannot be done without some kind of conditional surrounding the inflow rule that says, in effect, that it will not execute if the outflow rule can. While this will work, it is an unnecessary burden on the rule writer and will make the policies difficult to understand. This burden will increase if all four variables have to be accounted for. It will become overwhelming if any variables can be set using the rules.

This last point leads to the second issue that the USBR users wanted addressed. They wanted the ability to set any variable on an object, as opposed to being restricted to setting only the four variables mentioned previously, since this restriction precluded the creation of a large number of potential policies. The original motivation for this restriction was a desire to keep the *RSS* rule language simple and because the designers felt that no other slots needed to be set to model the Colorado River Basin. While this latter point is debatable – the USBR wanted to be able to set any slot – *RiverWare* had to be able to model any river basin and this kind of restriction would most likely make that impossible.

A last feature of the *RSS* policy language that the USBR decided they wanted changed was the manner in which rules were tied to specific objects and object types. Recall that the header line in the *RSS* rule language specification calls for a name, a variable and an object or class of objects. If the header contained the name of an object, the rule could only be executed by that object. If the header contained the name of a class of objects, it could be executed by all objects in that class. Although this

worked for the rules written for *RSS*, it seemed unnecessarily restrictive. Even so, it took a while for both the USBR users and me to convince ourselves that a rule should be able to set any slot. The creators of the *RSS* policy language were no longer available and there was no written record of their reasoning behind this restriction. Accordingly, it took the iterative process of writing the users' policy descriptions to come to the conclusion that this was not a required or even desirable part of the language.

3.5.4. Expert Systems

In addition to *RSS's* rule language, I also investigated different rule languages and expert systems. This was in part because I thought these could either be used as the basis for the new language or at least provide insight into what I might want to include in the new language. Another reason I initially looked into them was because *RSS's* language was referred to as a rule language and I wanted to understand what a rule language was.

3.5.4.1. General Literature

To start the process, I read about expert systems [Mettrey, 1991, Brownston, et al., 1985, Buchanan and Shortliffe, 1984, Neches, Swartout and Moore, 1984, Hayes-Roth, Waterman and Lenat, 1983], which provided valuable information about how they work and how they efficiently execute prioritized rules. One notable feature of many expert systems is the inclusion and maintenance of a list of dependencies for each rule. Dependencies are state variables that, when updated, cause the rule to be marked for re-execution. In the case of the new rule language, the dependencies would be the object variables that the rule might access during a given execution. If the rule had been executed and none of its dependencies had subsequently changed there would be no need to re-execute it, since its execution would produce the same results. On the other hand, if one or more of its dependencies had changed, it would be necessary to re-execute the rule, since its results might be different. It is also possible that the rule's re-execution would not alter its results, although there is no way to know this without re-executing the rule. Therefore, in order to insure that the rule's results are used by the simulation, the rule

must be re-executed. The gain associated with dependencies is execution efficiency in that rules are only executed when they have a chance of producing results that differ from their previous execution.

Another feature gleaned from my reading was the concept of an agenda. An agenda is a list of rules, ordered by priority, that are ready to be executed. These rules may either be rules that have never been executed or are rules whose dependencies have been updated since they were last executed. Either way, the agenda is used to quickly identify which rule should be executed next. Generally, this is the rule with the highest priority of all of those on the agenda. While neither of these features were strictly necessary in a rule language, they improved the efficiency of the rule execution mechanism.

3.5.4.2. CLIPS

I also looked into a number of expert system shells and languages. Most were proprietary systems that appeared able to be used as a basis for *RiverWare's* rule language. They included OPS83 [Neiman and Martin, 1986], Kappa [Kappa, 1993], Nexpert Object [Neuron, 1994], RAL [Forgy, 1994], and ART [Baum, 1993]. Unfortunately, only a subset of them were usable within *RiverWare* and they were ultimately rejected because of their cost. Any software that would be integrated into *RiverWare* would have to be free or very inexpensive. These systems were generally quite costly and some imposed individual license costs if they were embedded within a commercial product.

One of these systems, however, was freely available and looked like it could be used as the base rule language. CLIPS, short for C Language Integrated Production System [CLIPS, 1994, Giarratano, 1993], appeared to have the functionality necessary to both express the USBR users' policies and be integrated into *RiverWare*. It also appeared to address most of the deficiencies associated with the *RSS* rule language. It separated rules from functions, the rules were prioritized and it had local variables. In addition, since the source code was provided, it could be customized to fit the needs of the users. Its main drawback was that its syntax was LISP-based. In addition, it was not as flexible as a general purpose language.

I informed the USBR about this language and both the positive and negative issues regarding its use. They were wary of it, mostly because of its LISP-like syntax, although they agreed to keep an open mind about it and consider it if, on balance, it seemed like the best option.

3.5.5. Deadlines and Goals

For some time, the USBR had been hoping to meet a deadline wherein they would have all of the rules converted from *CRSS* into the new rule language. Initially, the extraction of the policies from *CRSS* consumed their energy and the need for a working system was not great. Yet, as the progress associated with the iteration toward complete policy descriptions began to wane, they began to realize that they needed a working language system in order to completely define their policies and meet their deadline.

While this was a reasonable need from their perspective, it presented a problem from mine. On the one hand, without a working system, they did not seem able to fully define their policy descriptions. On the other hand, without fully defined policy descriptions, it would be difficult to design a language that met their needs. Granted, I would never know the full extent of their policies in advance since a significant reason for wanting the policies external to *RiverWare* was so they could experiment with them. The problem was that given the small amount of time available to build a working system, an incomplete policy set would present a significant risk. For if the language designed to address the policies in their current state needed to change as the policies evolved, the users would be unlikely to meet their deadline. In addition, given the slow and uneven process of defining their policies, I was confident that their policies would require many changes before they were able to match *CRSS's* results.

One possible way out of this dilemma was to make the rule language either based on or similar to a general purpose language. Such a language could be enhanced with the functions necessary to allow the users' rules to communicate with *RiverWare*. This type of language could include the syntax necessary to represent their policies even with the inevitable changes that would be required. The main problem with this approach was that a general purpose language might not necessarily meet their long

term needs of a language that was easy to read, write and maintain for the many users who had little programming experience.

A number of public domain languages were available that could be used as a basis for the new language. All but one of these languages were general purpose scripting languages and all could be used with a relatively small amount of design and coding when compared to designing and coding a custom language.

Given the aggressive deadline and the fact that designing and developing a custom language would result in a significant risk of missing this deadline, the decision was made to use an existing language. This decision rested largely on the fact that the time anticipated to design and develop a custom language and then have the users define their policy descriptions using this language would exceed the amount of time available. In fact, it was determined that even if the first cut at the language was perfect, the time needed to design and develop it would leave the users with insufficient time to use the language within *RiverWare* to completely define their policies.

While this choice was by no means ideal, there were some positive aspects associated with it. The main one was that giving them a language quickly would allow them to fully specify their policy descriptions as rapidly as possible. In addition, a general purpose language would not require numerous changes as their policy descriptions evolved. Although there would need to be *RiverWare* interface routines added as the policies changed, these would be fairly minor additions and would be unlikely to significantly impact the users' schedule. Another benefit of such a system would be that I could invest relatively little time developing a system that would allow me to see their complete policies. Thus, if this system proved too general, I would have their problem descriptions available to design a better system. Lastly, it would give me a system on which I could do some concrete user testing and identify areas for improvement.

Another reason one of these languages seemed appealing was that as the policies were being defined, it was becoming increasingly obvious that a language that included general purpose constructs

was desirable. Admittedly, this was greatly influenced by the source of their policy definitions – *CRSS* FORTRAN – and the pseudo-code language given to them to refine their policy definitions. At the same time, these languages were the type with which they were most familiar and other language paradigms were likely to be difficult for them to learn. And since the early part of the language design was concerned with assisting them in their policy definition, it seemed better to give them a language that allowed them to focus on this task rather than making them spend what little time they had learning an unfamiliar language and trying to translate FORTRAN into it.

These benefits combined with the more important goal of getting their policies defined and used within *RiverWare* were enough to convince us that a language system needed to be developed as quickly as possible. All agreed that it should be textual, since this would take much less time to develop than a visual language. We also agreed that the language would be a temporary solution that would either be replaced or overlaid with another, probably visual, language.

3.5.6. Options Considered

There were a small number of languages that seemed to be usable as the basis for the rule language. These were Tcl [Ousterhout, 1994], Perl [Schwartz and Christiansen, 1997, Wall, Christiansen and Schwartz, 1996], Python [van Rossum, 1995], Java [Arnold, Gosling and Holmes, 2000, Cornell and Horstmann, 1996] and CLIPS. Each had strengths and weaknesses that will be discussed in the remainder of this section.

3.5.6.1. Tcl

Tcl is an interpreted scripting language that can be easily embedded into existing C applications. It is extensible in that new Tcl procedures can be defined in C. These procedures can be called from any Tcl script that is invoked from the C program. Depending on the Tcl procedures defined in the C program, the Tcl script can be used to perform any task that could be performed by a compiled C function. And, since Tcl scripts are defined outside of the C program, they can be changed without rebuilding the C program. These characteristics make it an ideal language to use when portions of an

application need to be defined outside the core executable. Since this is exactly what was needed as a basis of the policy language, Tcl seemed like a good choice.

Tcl also has some characteristics that are not ideal. For instance, Tcl's syntax is different from most procedural programming languages. Although it would be beyond the scope of this work to discuss all the differences, a discussion of a number of the more common differences will help to illustrate the point.

Figures 10 and 11 contain examples of some very simple Tcl and C statements. Each line has the same functionality in both Tcl and C. The first line, simple assignment of a number to a variable, illustrates one fundamental difference. Tcl uses prefix notation. Thus, the language requires the writer to call a function with two arguments to assign one value to a variable. The use of prefix notation using functions is a fundamental characteristic of Tcl and one that gives it a great deal of flexibility. For just as Tcl has defined a function called `set`, an application can define almost any Tcl function needed. This includes the redefinition of `set`, if desired, and the definition of functions to perform loops in addition to those provided by the language. For even Tcl's `while` loop is defined as a function.

```
set a 0
set b $a
set c [func 1 2 3]
set d [expr ($a + $b + $c) / 3]
```

Figure 10. Tcl source code example.

```
a = 0;
b = a;
c = func (1, 2, 3);
d = (a + b + c) / 3;
```

Figure 11. C source code example.

The second line shows how assignment of one variable's value to another variable is accomplished. Here, `b` is assigned `a`'s value, which was set to `0` in first line. Note the use of the dollar sign that precedes `a`. The dollar sign serves as a means of getting the value stored in the variable. Had it been omitted from `a`, `b` would have been assigned the string `"a"`. The need for the dollar sign, illustrates

another fundamental characteristic of Tcl. Everything in Tcl is represented as a string; even numbers. This characteristic is a common source of errors users new to Tcl.

The third line illustrates the manner in which functions are called in Tcl. Since everything is a string in Tcl, there needs to be a way to invoke a function. This is accomplished by using square brackets. Square brackets accomplish what is called command substitution and cause the text within the square brackets to be treated as a command to be executed. Thus, Tcl sees the word, `func`, as a function name that is called with 3 arguments. The result of this function invocation is assigned to the variable `c`. In addition, as might have been noticed in the first two lines, function arguments are space separated, rather than comma separated. In fact, a space is the way Tcl separates most functions from arguments and arguments from each other. The exceptions include the use of square brackets, as above. Everything within square brackets is considered a single entity. Thus `[func 1 2 3]` is considered one argument, rather than four.

The last line demonstrates how basic arithmetic is performed in Tcl. The function `expr` indicates that an arithmetic expression follows and needs to be evaluated. Note that the arithmetic expression is represented using infix notation, unlike the rest of Tcl. The ability to parse and evaluate infix notations is not part of the core Tcl language, per se. Rather it is embedded in the `expr` function.

In addition to its unusual syntax, a potential liability of Tcl is its relatively poor execution speed. This is in part because Tcl is interpreted, although other interpreted languages, such as Perl, are considerably faster. On the positive side, however, Tcl was being used in *RiverWare* as the basis for its model definition language, which is used to represent *RiverWare* models in files. The users frequently edited these files directly in order to make simple changes to models without incurring the overhead associated with starting *RiverWare*. On balance, Tcl was considered a reasonable option.

3.5.6.2. Perl

Perl is also an interpreted scripting language that was considered as the basis for the rule language. It is a very powerful language that can be used to quickly develop small to moderate sized

programs. It also can be extended to include user-defined functions. Perl is also very fast for an interpreted language. In many respects its syntax is C-like, although it contains many additional syntactic conventions. These include many shorthand notations, a larger number of operators and powerful regular expression and string manipulation capabilities. Perl's authors consider one of its strengths to be its flexibility. There are many ways to accomplish the same task and, in general, there is no prescribed way to do any particular task. For example, Figures 12-16 contain Perl code that echoes what is typed at the keyboard. All are considered correct and take advantage, to increasing degrees, of Perl's many shorthand notations. While these shorthand notations can make creating Perl programs relatively easy, they can also make them difficult to read. This is especially true for those who are not acquainted with Perl's idiosyncrasies. In fact, some have characterized Perl as a "write-only" language [Schwartz and Christiansen, 1997].

```
while ($line = <STDIN>) {
   print $line;
}
```

Figure 12. Perl example 1.

```
while (<STDIN>) {
   print $_;
}
```

Figure 13. Perl example 2.

```
while (<STDIN>) {
   print;
}
```

Figure 14. Perl example 3.

```
while (<>) {
   print $_;
}
```

Figure 15. Perl example 4.

Although this is certainly an oversimplification, it is true that Perl programs can be very cryptic and one could argue that this is not a language for those who are new to programming. Another

```
while (<>) {
    print;
}
```

Figure 16. Perl example 5.

characteristic of the version of Perl with which I was working was that it was difficult to embed into an existing application and then extend that language to allow for additional functionality. It was certainly possible, but not nearly as easy as with Tcl. In addition, the documentation for doing this consisted of the following line: “Look at perlmain.c and do like that.” Although I was able to follow this less than helpful advice and construct a sample program that embedded Perl and added a few sample routines, it seemed unnecessarily difficult. This alone, however, was not enough to persuade me to reject Perl. In the end, I rejected Perl because I felt its syntax would be very difficult to use and would encourage the users to write rule that were difficult to read and maintain. Since these were two important goals of the language, it seemed that using Perl presented an unnecessary risk.

3.5.6.3. Python

Another language option was Python. This language shares some characteristics of both Tcl and Perl. Like those languages, it is an interpreted scripting language. Like Tcl, it can be easily embedded and extended and like Perl it is quite efficient. Unlike either of them, however, it has a very structured syntax. In fact, a notable feature of Python is that it uses indentation to denote grouping within *if* and *while* loops. This feature, while considered overly structured by many, seemed to be a good one in terms of readability. In general, Python was a considerably more readable language than either Tcl or Perl.

Despite this, Python was also not chosen. This was because it was a relatively new language at the time I investigated it. As a result, books were not yet available for the language and it was unclear whether the language would gain a sufficiently large following to survive. I felt that any language I chose needed to be sufficiently well documented and accepted so that the users would not be taking any undo risk associated with using it.

3.5.6.4. Java

Java was a new, general purpose programming language at the time of this case study. It appeared to be able to provide the functionality necessary for the users' policies and would be reasonably familiar to the users. Unfortunately, at the time, there was no way to easily integrate it into a C or C++ program. These interfaces have since been added and had they been available at the time, I might have chosen Java. As it was, it could not be used.

3.5.6.5. CLIPS

CLIPS was the last language considered. It was unlike the other languages in that it was specifically a rule language that included a rule processor for executing rules. Like Tcl, Perl and Python, it could be embedded into an existing C application. These two characteristics made it a strong candidate for use as a basis for the policy language. Its main drawback was that it had a LISP-like syntax. Like Tcl, it was also considered a viable option.

3.5.7. Option Selected: Tcl

In order to come to a decision about the language to use, I decided I needed to get feedback from the USBR. To this end, I wrote a policy, *Upper Basin Rule Curve*, using Tcl and CLIPS. In addition, I included two hypothetical languages that showed them what might be possible given additional development time. Appendix D contains these examples. The Tcl and CLIPS examples were provided as near term solutions and, as discussed above, were the best options given the users' tight schedule. The additional examples were provided as longer term solutions and I did not expect either of them to be selected. I presented what I considered the positive and negative aspects associated with each option and asked for their opinion.

They unanimously chose Tcl. This choice was predicated largely on the fact that they were very familiar with it and felt that they could quickly write their policies using it. Since time was of the essence, this seemed reasonable. They also knew that I could quickly develop a language using Tcl. Although they were concerned with Tcl's performance, they felt that this solution was short-term and if

Tcl turned out to be too slow, it could be replaced. CLIPS was rejected almost entirely on its LISP-like syntax. This was true even given the fact that, as a rule language, it contained many of the features that would have to be built into Tcl.

3.5.8. Development

The decision to use Tcl meant that the core language design and development were taken care of. Yet, there were a number of other areas that needed to be designed and developed in order to have a complete rule language system. I was aware of a number of these components and what was needed from a high level, although I would learn more as the design and development progressed. Those areas about which I was confident included extensions to Tcl to support the naming, prioritization and dependency specification of rules and a number of functions needed to allow the rules to both read and write the values of *RiverWare* state variables. I also needed to include within *RiverWare* the ability to load and store the rules and to execute them when requested. In addition, I needed a rule-based simulation controller that would drive the simulation of *RiverWare* and request rule executions when necessary. Lastly, I needed to provide a GUI-based programming environment to support the users in their use of the rules. Although I was not sure exactly what this environment would look like, I knew that I needed at least a means of loading rules into *RiverWare* and a means of editing the rules.

This stage of the development progressed fairly quickly. I was able to develop a working system in a few months that allowed the users to start writing and using their rules with the new language system. This section will provide a brief overview of the development process. Section 3.6 will discuss the resulting language system in detail. Although, the development progressed in all areas simultaneously, the discussion of the development will be broken into three areas: the programming language, the programming environment and the language processor.

3.5.8.1. Language

As stated above, Tcl provided the core language around which the rule language would be based. It provided most, if not all of the syntax needed to define the logic of the policies. This helped

greatly in terms of design and development time, although the language needed to be augmented in order to allow for the functionality necessary to support a rule language. As a general purpose scripting language Tcl did not have any built-in facility to support naming or prioritizing rules, dependency specification, or unit checking and conversion. With the exception of unit checking and conversion, however, they could be added fairly readily.

Based on the feedback I had received from the USBR, I decided I wanted the main structure of the language to look like the Tcl example I had given them, since they had liked it. Refer to Appendix D for the example given to the users.

In addition to the overall syntax of the rule, I also needed to write a library of routines that could be used to access values in a *RiverWare* model. These generally amounted to routines to read and write values on object variables. It also included print statements that could be used to track the progress of the rules as they executed. This latter feature was originally implemented to allow me to insure that the rules were executing as planned and evolved into a frequently used rule tracing facility.

3.5.8.2. Programming Environment

Once the language was designed and much of it coded, I began work on the programming environment. Although I knew I needed a programming environment and knew many of the facilities that the USBR wanted, I had no clear design specified in advance. As suggested earlier, this was a mistake. The programming environment design turned out to be driven in large part by a notion of what the users wanted and the facilities that I needed to test my code. In general, where the needs of the users were emphasized, the programming environment was helpful and where my needs were emphasized, the programming environment was, at best, marginally helpful.

The basic functionality provided by the programming environment was the ability to load text files into *RiverWare*, to view the results of these loads, to edit the loaded rules and to view the output of user-defined print statements as the rules executed. These capabilities were generally useful to the users, although they were incomplete. For instance, I did not provide a means of saving changes made to the

rules from the GUI. This was largely due to time constraints and the fact that the users decided that other functionality was more important.

Another prominent feature of the programming environment was the fact that it relied heavily on user-managed UNIX files. As a result, the users needed to be familiar with the UNIX file system and at least one UNIX-based editor. While this was true for many users, there were enough for whom it was not true that requiring the users to rely on user-managed UNIX files was not the best solution.

3.5.8.3. Language Processor

There were two main tasks that needed to be addressed with respect to the language processor. The first was the need to construct a rule-based simulation engine within *RiverWare*. As discussed earlier, this engine would be similar to *RiverWare's* simple simulation engine. It would differ in its ability to request execution of rules and to use the results of these executions to solve the river system. This part of the development was assigned to another developer and I had only minor input into the design and development.

The second task was the need to construct the facility to load and execute the rules. The ability to load rules was largely accomplished while writing the language. The ability to execute rules was only partially accomplished while writing the language. The parts that were accomplished included the ability to execute the rules and the ability to read and write values in a *RiverWare* model. The part that was not accomplished involved the mechanism necessary to execute the proper rule when the rule-based simulator asked for a rule to be executed. This involved the creation and maintenance of an agenda and the integration of rule dependencies into the system.

As discussed earlier, the agenda is a list of rules, ordered by priority, that are available for execution. The agenda only contains those rules that must be executed. The determination of which rules must be executed is based on whether or not their execution can change the state of any objects in a *RiverWare* model. Therefore, every rule is placed on the agenda at the beginning of each of the simulation's time steps. This is because there is no way of knowing whether or not a rule's execution

can change the state of *RiverWare* at the start of a time step and it must be placed on the agenda in case it can. After a rule has executed once for a given time step, however, it can be determined whether or not it makes sense to re-execute it by using its dependencies, as will be explained next. Accordingly, after a rule is executed, it is removed from the agenda and only returned during the current time step if there has been a change made to any of its dependencies.

A rule's dependencies are the values that a rule reads or writes during its execution. A rule can have any number of dependencies. If one or more of its dependencies' values changes after a rule has executed, it is possible that the rule will follow a different execution path and produce different results if it is re-executed. In order to insure that the policy that the rule represents is adhered to, it needs to be executed if any of its dependencies changes. In this way, whatever value or values the rule produces will be based on the latest state of the system. For *RiverWare*, a rule's dependencies were the values stored on objects in a *RiverWare* model. These values were stored in variables on the objects called slots and were referred to as *object.slot*s.

The rule in Figure 17 will be used to illustrate how dependencies work in a simple rule. This rule, named `MeadStorageControl`, is used to insure that Mead's storage (i.e., `Mead.Storage`) will never be greater than 300,000 acre-feet. Its only dependency is `Mead.Storage`, since that is the only value that is either read or written by the rule. If, during a simulation, another rule or a simulation method sets `Mead.Storage`, `MeadStorageControl` will be placed on the agenda and eventually executed. When executed, it will check the value of `Mead.Storage` and, if it is greater than 300,000 acre-feet, it will reset it to 300,000 acre-feet. If the value of `Mead.Storage` is less than or equal to 300,000 acre-feet, the rule will not make any changes. If `Mead.Storage` is never changed, it will not be placed on the agenda until the next time step. Thus, the value of `Mead.Storage` will be constrained to a maximum of 300,000 acre-feet and `MeadStorageControl` will only be called if its execution can be of use.

```
RULE_NAME: MeadStorageControl
RULE_DEPENDENCIES: Mead.Storage
BEGIN
   if {Mead.Storage > 300,000 af} {
      Mead.Storage = 300,000 af
   }
END
```

Figure 17. Simple rule with a single dependency.

3.5.9. Refinement

Once I had the basic functionality coded, the users started to write their rules using the language. Initially, I did not have the language system integrated into *RiverWare*, although this did not matter to them since they were in the early stages of their rule writing. Once the rule-based simulation engine was complete, I integrated the language system into *RiverWare* and the users were able to start running some simple rule-based simulations using the language.

This period can be thought of as a refinement period for both the users and the developers. This is largely due to the fact that the code for the rule-based simulation engine and the rule language system were new and untested by actual users. It was also due to the fact that the users were using this system to define their policies. As discussed early, they needed a working system and the ability to generate actual numbers in order to be confident that their policies were properly defined. As it turned out, their policy definitions changed considerably during this process. For the most part, their policy definitions turned out to be far more complex than they originally thought they would be.

During this period, I was essentially on call to help with any problems that the users had with the rule language system. This included being available to add Tcl functions to the language, fix bugs in the language, the programming environment or the language processor. There were actually few bugs in the language and the processor; most of the bugs and enhancement requests related to the programming environment. This is partially because of the nature of user interface programming in that users are always able to find ways to use a system in ways that were not intended by the programmers. It was also due to the fact that the programming environment was not designed properly in advance and changes needed to be made to accommodate unanticipated user needs.

This period consumed a great deal of time largely because of the difficulty that the users had defining their policies. This was primarily due to the fact that they were using the policies contained within the *CRSS* code as the basis for these policies. As mentioned earlier, *CRSS* was full of bugs and the policy code was tightly coupled with the simulation code and with the code of separate policies. As a result, it took a great deal of time to extract the policies and write them using the new language. In addition, as bugs or logical problems were encountered, the users felt that they needed to fix them in order to allow them to make an accurate comparison of *CRSS* and *RiverWare* once the policies were fully defined.

Given the users' need to produce a full set of rules to match *CRSS's* results, I was tasked with helping them define their rules. In part, this was due to the fact that they were novice Tcl programmers and needed help with its syntax, although this need waned rapidly. It was also in part because of their propensity to design their policies on the fly. They often coded whatever gave them the results they wanted, regardless of whether or not this resulted in readable, maintainable or, as will be discussed later, correct rules.

In addition to the help I could provide regarding their rules, this role also had a number of unanticipated benefits. The first was to have access to the users' problem descriptions as they evolved. This gave me the ability to see where additional functions would make their rule specification more readable and easier to write. Generally, these functions were not requested. An example of one of the functions is the summing of a series of object.slot values over a user-defined date range. The other, less tangible benefit, at least from the users' perspective, was insight into their programming practices. While this will be discussed at length in the next chapter, it became apparent that the flexibility offered by this general purpose language was not ideal for the users. On the one hand, some of the users were comfortable with the generality provided by the language and tended to program whatever was necessary to achieve the results they thought were correct. This included writing code that was difficult to maintain and code that omitted elements that were necessary to insure correct results. On the other

hand, there were those who had a great deal of difficulty using a general purpose procedural language. Generally, these individuals had little experience as programmers.

The main drawback to the refinement period was the amount of time it took. Given the complexity of extracting and defining their policies, the users were not able to meet their deadline. In fact, it took them a considerable amount of time before they were able to demonstrate that *RiverWare* and their rules could match the results of *CRSS*. In this time, they became very proficient in the use of the rule language and, generally speaking, liked it. The main problem with the length of time it took the users to define their policies related to the fact that this language was originally intended to be a short-term solution that would be replaced by a language that more closely approximated their needs. Yet, given the amount of time it took to define their policies, work did not start on the new language until much later than anticipated.

3.6. User Test

In June, 1995, the USBR brought in a set of users from the Colorado River Modeling User Involvement Group (CRMUIG). These users, as described in Section 3.2.2, would constitute a significant portion of the early user population. They had a wide range of programming experience, which ranged from novice programmers with little to no experience to a few with over fifteen years of FORTRAN experience. This user test was conducted as part of a three day training class. The users were new to *RiverWare*, although they were familiar with the functionality that it needed to provide. Given that they were new to *RiverWare*, they also knew nothing about the rule system.

The main intent of the class was to teach them how to use *RiverWare* and the rule system. Therefore, they were given talks and tutorials that taught them many of the basic features of *RiverWare*, including its GUI. They were also given talks and tutorials about the rule system.

The rule system tutorial relied on a simple *RiverWare* model and ruleset very similar to the one shown in Figures 9a-9i in Chapter 2. This model and ruleset were given to them completely specified and they were asked to run simulations using it. This part of the exercise was intended to teach them the

basics of loading and using rulesets. It was also intended to teach them how to do simple, non-code-related manipulations of the rule system. This included changing priorities and viewing diagnostic messages generated by the rules during simulations. They were also asked to view the states of objects before and after a rule-based simulation in order to see the effect that the rules had on *RiverWare's* objects.

Once they had mastered this, they were asked to make changes to the rules. This included changing numbers that were either specified in the rules or specified on objects referenced by the rules. After each change, they were to run the simulation and view the effect the changes had on the *RiverWare* objects. In this way, they could learn how to manipulate the rules. Changing the rules themselves required that they use a UNIX editor. Given that there was a simple GUI-based editor available, this was not too difficult, although some had some trouble with it.

Lastly, they were asked to create their own ruleset and rules from scratch. This required them either to create new files and enter the text necessary to define a ruleset and rules or to copy existing ruleset and rule files and use the copies as the basis for their new ruleset and rules. This was difficult for those who were not familiar with UNIX and did not know how to create or copy files using it. This turned out to be a significant shortcoming of the programming environment.

In addition to this shortcoming, some of the users were new to programming and had fundamental misunderstandings about what could and could not be done using the rules. For example, one user read a value from a *RiverWare* object.slot and assigned it to a local variable. He then changed the value of the local variable and thought the original *RiverWare* object.slot value was also changed.

Despite this and the above problems, however, the users found the use of rules within *RiverWare* to be useful and reasonably intuitive. This may largely be based on the fact that they had never used a system remotely like this and were willing to forgive the language system's shortcomings in order to take advantage of the power it provided.

Overall, this user test convinced me that the rule language system could not work for the intended target audience of non- to novice programmers. For one thing, a language that required less programming skill was necessary. A general purpose programming language required more programming background than the users were likely to have. In addition, an environment that provided better support was also necessary. A fuller discussion of these issues will be presented in the next chapter.

3.7. Final Product Description

This section will describe the first rule system in its final form. It will be broken into subsections that will describe the programming language, the programming environment and the language processor. This section will concentrate on the system itself and will not delve into many of the details of the users' experience and in what ways the system was good and bad. This information will be discussed in the following chapter.

3.7.1. Language

As discussed previously, the language was a Tcl-based textual language. The main parts of the language were a ruleset, rules that were part of a ruleset and a set of Tcl functions that were used by the rules and by each other. Each of these areas will be described in turn.

3.7.1.1. Rulesets

A ruleset was a file that contained references to rule files. These references were file names that could be preceded by an optional relative or absolute path. A ruleset was intended to be a grouping of related rules that formed a single coherent policy set, although a ruleset could contain any rules that the user wished to place in the ruleset. A ruleset could be used by one or more *RiverWare* models during rule-based simulation runs. Figure 18 shows a sample ruleset that contains references to seven rules. Note that the rule files could be stored anywhere that was accessible by the ruleset file. This addressed a user request to be able to store rule files in multiple directories. One reason for this was to store sets of related rules in their own directories.

```
MeadRuleCurve.tcl
MeadMinRelease.tcl
PowellForecastError.tcl
../MeadFloodControl/SetMeadOutflow.tcl
../MeadFloodControl/SetSNWPSchedule.tcl
../MeadFloodControl/SetCAPSchedule.tcl
../MeadFloodControl/SetMWDSchedule.tcl
```

Figure 18. Sample ruleset.

3.7.1.2. Rules

Rules were the main policy specification code. They were stored in individual files and contained the information necessary to define a particular aspect of a policy. Generally speaking, a number of rules were combined to form an overall policy definition, as described above. A template of a rule is shown in Figure 19. It contains a number of placeholders surrounded by angle brackets. These placeholders had to be filled in by the user. The words in all capitals were keywords and had to be specified as shown.

```
RULE_NAME: <rule name>
RULE_PRIORITY: <rule priority>
RULE_DEPENDENCIES: <rule dependencies>
INCLUDE
  <rule includes>
BEGIN
  <rule body>
END
```

Figure 19. Tcl-based rule template.

The first placeholder, `<rule name>`, was used to specify the name of the rule. This name had to be unique with respect to all other loaded rules. It could include most characters, including blanks, although leading and trailing blanks were stripped. As a result, it had to contain at least one non-blank character. The next placeholder, `<rule priority>`, was used to specify the priority of the rule. The priority had to be a positive integer that was unique with respect to all other loaded rules.

`<rule dependencies>` was used to specify a space separated list of the rule's dependencies. Any number of dependencies could be specified. The form of a dependency was `<object>.<slot>`, where `<object>` was the name of an object in a *RiverWare* model and `<slot>` was the name of one of

the object's slots. If either the object name or the slot name contained a blank character, the entire dependency name had to be quoted. Although the intent of the dependencies was to list those object.slots that could be either read or written during the rule's execution, this restriction was not enforced. This was because it would be difficult, if not impossible, to statically determine a rule's dependencies. One reason was that Tcl allowed for object.slot names to be generated at run time by combining strings. The other was that Tcl supported arbitrarily deep nesting of conditionals.

`<rule includes>` was used to place Tcl commands that would include, or "source," external files that contained Tcl code. These files, and indeed this section itself, could contain any Tcl code. This section was intended to support the inclusion of Tcl procedures that were defined in external files and used by the rule. These procedures could be commonly used procedures or procedures that were only used by a single rule. This section had a number of liabilities associated with its use that were related to Tcl. The first related to the fact that Tcl had only one namespace. As a result, any procedure that was declared in this section was visible to all rules, even if it was only intended to be used by one. Generally, this was not a problem, although obscure errors could result if one rule inadvertently called a procedure intended for use by another rule. Another problem related to Tcl's single namespace was how duplicate procedure names were dealt with. If two procedures with the same name were declared, the first procedure's definition would be overwritten by the second. This could lead to errors if it was not recognized. To mitigate this somewhat, the programming environment warned the user when a ruleset was loaded with duplicate procedure declarations.

`<rule body>` was used for Tcl commands that formed the body of the rule. As with the include section, any Tcl code could be placed here. Generally, this section contained the high-level logic that defined the rule and determined whether or not one or more *RiverWare* object.slots needed to be assigned new values and, if so, what those values they should be. Of course, the rule writer was free to structure the code as desired.

3.7.1.3. Procedures

In order to support the definition and use of procedures that were independent of rules, the rule language allowed Tcl procedures to be defined in files that were external to rule files. As discussed above, these files could be included by one or more rules and their procedures used by the rule's Tcl code. These procedures could also be used by other procedures declared in other included files. Figure 20 contains an example of a procedure used to compute the amount of water consumed above a specified reservoir.

```
proc ConsumptionAbove {res} {

  set date [C_GetDateTime]

  if {$res == "Navajo"} {

    set nav [C_GetValue NavIrrigation.TotalDiversionRequested \
             1.0 acre-feet/month $date]
    return $nav

  } else {
    return 0.0
  }
}
```

Figure 20. Sample Tcl procedure.

3.7.1.4. Predefined Tcl Routines

A number of predefined Tcl routines were made available to allow the rule writers to define their policies. Examples of the use of some of these routines are shown in Figure 21 and will be accompanied by brief descriptions that should give a sense of how they were used. `C_GetValue` and `C_SetValue` were used to read and write object.slot values in a *RiverWare* model for May, 1996 and May, 1994, respectively. `C_Print` was used to display user-defined output as rules executed. `C_SolveStorage` returned Mead's storage in acre-feet given an inflow of 125.0 acre-feet/month and an outflow of 118.0 acre-feet/month. Note that Mead's previous storage was not specified, which meant that the current value of Mead's previous storage was used. Lastly, `C_ElevationToStorage` returned Mohave's storage in acre-feet given an elevation of 45.0 meters.

```
C_GetValue Powell.Storage 1.0 acre-feet “May 1996”
C_SetValue Mead.Outflow 1.0 acre-feet/month “May 1994” 798783.0

C_Print “mead_outflow: $mead_outflow”

C_SolveStorage Mead 125.0 1.0 af/m 118.0 1.0 af/m 1.0 acre-feet
C_ElevationToStorage Mohave 45.0 1.0 m 1.0 acre-feet
```

Figure 21. Sample predefined Tcl routines.

The following sections will present a brief overview of the predefined Tcl routines. They are grouped for convenience of explanation. For a complete list of these routines, see Appendix E.

Value Access Routines. These routines gave the rule writer access to the object.slot values in a *RiverWare* model. Routines were available to read both single and multiple values. In case of multiple values, they had to be for a single object and within a contiguous date/time range and could be returned as a list or as a summation. Routines were also available to write both single and multiple values. There were also routines that provided the ability to copy the value(s) from one object.slot to another and to constrain the value of an object.slot. Lastly, routines were available to test object.slots and values to determine if they contained usable information. With the exception of the test routines, each of these procedures took a scale and a unit. For procedures that read data, these were the scale and units in which the value(s) would be returned. For procedures that wrote data, these were the scale and units that the new value(s) were assumed to be in. These procedures also took an optional date/time. If specified, the date/time denoted the time step of the value to be read or written. If unspecified, the value was read or written at the current date/time in which the rule was executed.

Object-specific Routines. These routines were used to allow rule writers to perform mass balance calculations within their rules. The mass balances solved for inflow, outflow, storage and diversion and made use of the same routines that were used by *RiverWare* during a simulation. These routines did not make changes to *RiverWare* objects. Instead they were used to set the values of local variables within a rule so that the rule could determine the possible effect of an action.

Sub-basin Routines. These routines were used to access information about sub-basins stored in a *RiverWare* model. Sub-basins were groups of objects that were generally physically linked, although this was not required. An example of a sub-basin on the Colorado River was the Upper Colorado that was defined as all objects upstream and including Lake Powell. These routines were used to gather aggregate information on the objects in a sub-basin. The aggregations allowed included sum, average, minimum and maximum. An example of a typical sub-basin aggregation was the average inflow for all objects in the sub-basin for a single time step.

Conversion Routines. These routines enabled a rule writer to convert values from one scale and unit to another. They also could be used to perform reservoir-specific conversions. For example, one routine allowed for the conversion between elevation and storage on a specified reservoir. Lastly, there were routines that allowed rule writers to convert between flow and volume.

Date and Time Routines. The date and time routines were used to access date/time information on *RiverWare* related to the current simulation. Examples included routines to get the current date/time or the start or end date/time of the simulation. In addition to the entire date/time, the rule writer could also get individual components, such as a month or year. There were also routines to manipulate these dates and times. For instance, the rule writer was provided with routines to increment and decrement dates and times by components ranging from seconds to years. In addition, there were miscellaneous routines to get the days in the month and the number of seconds in the current time step.

Utility Routines. The remaining routines can be grouped into what can be called utility routines. These included fairly standard routines to return the minimum and maximum of two or more values. They also included routines to compute *CRSS*-specific values needed to show that *RiverWare* could match *CRSS's* results. There was a routine to stop the simulation if the rule determined it should. Lastly, there were two routines that allowed the user to display information about the execution of rules. These routines were essentially print statements that could be embedded within the rule's Tcl code.

3.7.2. Programming Environment

The rule system programming environment was a GUI-based set of tools that provided the functionality necessary to load, view, update and debug rules to be used by *RiverWare*. Although it was not a complete environment in that it relied on UNIX files and editors, the USBR users were able to use it to create and test rules that matched *CRSS* functionality. The following sections will discuss the functionality provided for loading and viewing rulesets and for rule execution tracing.

3.7.2.1. Loading Rulesets

A ruleset must be loaded into *RiverWare* in order for its rules to be used by *RiverWare's* rule-based simulation. This was accomplished from the main Ruleset dialog by using a file chooser to select a ruleset file. As the ruleset file was loaded, the user was presented with feedback on the status of the load. If there was an error, the loading was terminated and the user was given one or more diagnostic lines. The diagnostic lines contained both the problem and the line number. In addition, the diagnostic lines were linked to the file associated with the diagnostic and the user could open this file to the indicated line number by double-clicking on the diagnostic line. The file was placed in a user-defined UNIX-based editor. Note that in this case and in all subsequent cases in which editing was allowed, any changed files had to be reloaded in order to be used by *RiverWare*. In addition to loading rulesets, the user could also remove a ruleset from *RiverWare* by clicking on the "Clear" menu item.

3.7.2.2. Viewing Rulesets

Once loaded, the ruleset could be displayed to show which rules and procedures were included. There were four different views that were presented, which included the hierarchical structure of the loaded ruleset, a list of the loaded rules, the ruleset's agenda and the rules' dependencies.

The hierarchical view of the loaded ruleset contained a treeview representation of the ruleset, starting with the ruleset file and branching at each rule file. For each additional file included in the rule file, the user was able to drill down and view any files that had been loaded. This dialog did not permit editing of the ruleset or its structure. It did, however, allow the user to edit any listed file by double-

clicking on the line for that file. As with the ruleset loading diagnostic line, a user-defined UNIX-based editor was invoked.

The list of loaded rules displayed each rule in priority order. Figure 22 shows a populated dialog. Each rule contained a place for a check mark, which indicated whether or not the rule was active, a priority and a name. This dialog allowed the user to edit its contents. The active check mark could be toggled and both the priority and name could be clicked on and edited. Note the small icon with a red “R” in it. This stood for “Rule” and also allowed the user to invoke an editor by double clicking on the icon. Note too that any changes made to this dialog were not reflected in the original rule files and could not be saved there using the programming environment.

Figure 22. Edit rules.

The agenda dialog showed the current state of the ruleset’s agenda. It consisted of two lists. The left list contained those rules that were on the agenda and ready for execution. The right list contained those rules that were off the agenda and did not need to be executed. Both lists were in priority order. As the simulation progressed, the contents of the two lists were updated. Neither of these

lists were directly editable, although the user could reset the entire agenda. In addition, the list contents could be double-clicked to open the corresponding rule file.

The dependencies dialog was used to map the dependencies back to the rules that used them. Its contents could be used to determine what effect changes to *RiverWare* objects might have on the ruleset and its agenda. This dialog consisted of a single treeview with the rules' dependencies at the top level. Underneath each dependency was the name of one of more rules. As with the agenda dialog, this dialog was not directly editable and allowed the user to double-click on the rule name to edit the rule file.

3.7.2.3. Rule Execution Tracing

Rule execution tracing was a feature that allowed rule writers to add diagnostic messages to their rules and procedures and have those messages displayed to a dialog during a rule-based simulation run. It included the ability to select which diagnostic messages were displayed and to set breakpoints during the simulation. An example of a rule execution tracing dialog is shown in Figure 23. Note that the lines in black text were generated automatically by the system, lines in blue text were user-defined messages and lines in red (not shown) were system generated error messages. As with other parts of the programming environment, errors were associated with a rule or procedure and the user was able to edit the associated file directly from this dialog.

The users had some flexibility in deciding which diagnostic messages were displayed. They could place messages directly in their rule or procedure code. These would be displayed by default. The user could also turn on or off diagnostic messages by rule, by date and time, and by message type/level. Message types/levels were system-defined groups of messages that related to certain aspects of the rule-based simulation and rule execution. They were ordered such that each succeeding level contained the messages from the previous level. Figure 24 shows the tracing definition dialog with the message type/level pop-up menu open.

Figure 23. Rule execution tracing.

Figure 24. Rule execution tracing definition.

Rule tracing breakpoints gave the user the ability to halt the execution of the rule-based simulator and view the state of the system. No changes to the system were allowed while halted, however, as this would seriously compromise the state of *RiverWare*. Breakpoints were specified

externally to rules, although they were associated with rules. Any number of breakpoints could be specified and could be enabled or disabled.

Breakpoints had to be associated with a specific rule and could be given a date/time range within which they applied. In addition, the user could specify whether they wanted the execution to halt before the rule executed, after the rule executed or both before and after the rule executed. When a breakpoint was hit, the simulation's execution would stop and the user would be informed. At this point, the user could choose to continue execution; skip the execution of the rule, if the breakpoint was hit before the rule executed; or halt the simulation entirely. Before making this choice, however, the user could examine the state of the *RiverWare* objects.

3.7.3. Language Processor

The language processor was internal software responsible for loading and executing rules. The following section will briefly describe each of these functions.

3.7.3.1. Loading

Rulesets were selected for loading using the programming environment, as discussed earlier. When a ruleset was loaded, each rule was parsed in order extracting the name, priority, dependencies and body. During the loading, the included files were added to Tcl's list of procedures so that they could be used during execution. The rule's name and priority were checked to insure that they were unique and the dependencies were checked to insure that they existed in a *RiverWare* model. The latter was necessary in order to register the dependencies with *RiverWare*. This allowed the objects to alert the rule system to changes in the dependencies' values. In order to facilitate rule execution, the rule body was converted into a Tcl procedure.

3.7.3.2. Execution

The language processor was also responsible for the execution of the rules. Since the rules were written in Tcl, the execution of the rules was fairly trivial. It consisted of calling the rule's Tcl

procedure that was created at load time and setting up the facility to catch any errors. The bulk of the work associated with rule execution happened before a rule was executed. This work involved tracking the dependencies and keeping the agenda up-to-date. Tracking dependencies was fairly simple and involved catching the alerts sent by the objects when a dependency's value was changed. When a dependency's value changed, all rules that listed it as a dependency were placed on the agenda. The agenda was always maintained in priority order and, when requested, the highest priority rule was executed.

Chapter 4

Case Study 1 Lessons

4.1. Introduction

The *RiverWare* rule language system as described so far was partially successful. It provided the USBR users with a complete system that they were able to use to develop their rules. These rules were in turn used to demonstrate that *RiverWare* could duplicate *CRSS's* results, which was the USBR users' primary short-term goal. In addition, the USBR users liked the rule system and found that its power and flexibility contributed to their ability to develop and tune their rules. This is not to say that they did not have issues with the rule system, although they felt they could work around them. For non- to novice programmers, however, the rule system was not as successful. Many found it nonintuitive and difficult to use. They had problems with both the language and the environment. Since this type of user constituted the bulk of the eventual users, it was clear that this system, regardless of the USBR's opinions and experiences, did not meet the needs of users in general. In addition, even the USBR users who liked the system and had sufficient programming experience to use it properly, often did not use it as intended.

This chapter will examine some of the decisions made during the language's design and development and elaborate on those that were good and those that were bad. Individual decisions will not be discussed chronologically. Rather, the issues surrounding these decisions will be grouped and discussed. The topics that follow are the components associated with an application-specific language, the use of user provided problem descriptions, the users and the goals for the language.

4.2. Programming System Components

The components that will generally make up an application-specific language system are not initially obvious. I knew that a language was necessary and I knew that this language would somehow

have to be connected to the application. Other than that, I did not have a clear idea of where one part left off and another began. As the design and development progressed, it became clear that a number of fairly distinct components were necessary. The following sections will discuss each of these components, which include the programming language, the suite of subroutines that provide application- and domain-specific functionality, the programming environment and the language processor. Although there is some overlap between these areas, they provide a good categorization of the main areas that must be considered in the design of an application-specific language system.

4.2.1. Programming Language

The programming language is perhaps the most obvious component of an application-specific language system. It provides the basis for expressing the users' problems. As might be expected, there are a great number of programming language choices. Although it is beyond the scope of this dissertation to detail all possible options, this section will discuss some of the major decisions that must be considered when selecting a language.

4.2.1.1. Language Source

There are two main language sources that can be considered when designing an application-specific language. One is to design and develop a custom language. The other is to embed and extend an existing language. This section will discuss some of the advantages and disadvantages associated with designing and developing a custom language and with using and extending an existing language.

4.2.1.1.1. Custom Language

A custom language, as the name implies, is tailor-made to address user problems. Accordingly, it can be made to closely match how users express their problems. It can include data structures, functions and control structures that are particular to the application in which the language will be used. This can result in a language in which it is easy for users to create, read and maintain their programs. In short, a custom language can provide users with a language that matches their needs.

There are a number of drawbacks associated with custom languages. One is that they can be difficult to design and develop. The degree of difficulty will depend in part on the developer's familiarity with the process and tools associated with language specification and in part on the complexity of the language. Yet, even if the developer has experience with this process, a custom language will likely take more time to design and develop than using an existing language, which can be problematic if the language needs to be available quickly. Lastly, a custom language, which is likely to be task-specific, can be less flexible than an existing language. This can be a problem if the users' problems turn out to be more complex or significantly different than originally specified and can result in changes to, or even a redesign of, the custom language.

Initially, I assumed that I would design and develop a custom language for the *RiverWare* rule system. Although I had no experience doing so, it seemed like this option would best meet the users' needs. This notion began to change when it became clear that the USBR users were unable to specify their problem descriptions outside the context of an existing language system that could be used to generate results when used within *RiverWare* simulations. This meant that any custom language I developed, short of a general purpose language, would stand a good chance of being inadequate as the USBR users used it to converge on their rule definitions. In addition, the USBR users' rather aggressive deadline to have their *CRSS* rules converted to *RiverWare* dictated the rapid development of a language. Given this combination, I decided to consider using an existing language.

4.2.1.1.2. Existing Language

For the purposes of this discussion, existing languages are languages that have been designed and developed to be embedded into an existing application. They can be either free or commercially available products. In addition, they may be general purpose or domain-specific languages. One of the main characteristic that differentiates a custom language from an existing language is the fact that existing languages are typically more general purpose than custom languages. This is even true of rule languages, which need to allow for a wide variety of rules across multiple domains. Another characteristic, as mentioned above, is their ability to be embedded within an existing application. This is

important since it allows these languages to be used as a basis for an application-specific language. Lastly, existing languages can be extended to include some, if not all, of the functions and features that are needed to represent the users' problems.

Using an existing language can provide a number of benefits to the application-specific language designer. A major benefit is time savings, since the existing language will form the basis for the application-specific language. As a result, it will include the underlying syntax, data types and control structures. In addition, these languages will probably include many convenience features, such as input and output functions and perhaps even a programming environment. Another benefit is that the use of an existing language requires less expertise than designing and developing a custom language. This can be particularly important if the designer has little experience with the language design process or the tools used to create languages.

The benefits associated with existing languages do not come for free, of course. The language must be extended to include the necessary functions that will allow the resultant programs to communicate with the application. While this is a fairly minor price to pay, there are other potentially more costly issues. One is that if the language is a commercial product, any time saved in the design and development of the language may be offset by the cost of the language. In addition, the fact that these languages are more general purpose than custom languages may make them more difficult to use for inexperienced programmers. Lastly, these languages may be syntactically unappealing to users.

As mentioned in the previous section, the USBR users' need to have an existing language system to fully specify their rules, my lack of experience with designing and developing custom languages and the USBR users' aggressive schedule pushed me to consider using an existing language. A number of options were considered including public domain scripting languages, such as Tcl, Perl and Python, and both public domain and commercial rule languages, such as CLIPS. Cost, both initial purchase price and on-going license fees, dropped commercial products from consideration. This left the public domain products and, as was discussed in the previous chapter, the ultimate selection of Tcl.

4.2.1.2. Language Characteristics

Programming languages can have many characteristics that define how they can be used and how users perceive them. Although there are undoubtedly a large number of ways to categorize programming language characteristics, this section will break them into four sections. The first is the scope of the language, which indicates how general purpose versus task-specific it is. The second will be termed the language's style. Although this is a fairly vague term, it is meant to indicate where the language falls on the text to visual scale. The next is the language paradigm and this refers to whether the language is a procedural, functional or some other type of language. The last is a bit of a catch all and it discusses some language features that were of interest during the design of the language system.

4.2.1.2.1. Language Scope

Programming languages can be primarily general purpose, primarily task-specific or a combination of the two. A general purpose language is one that can be used to write most any type of program. While it may not be the best language for every type of problem, it is generally expressive enough to allow a programmer to create a program that solves his/her problem. A task-specific programming language, on the other hand, is a language that is designed to solve certain types of problems. These problems may be from either a particular domain or may be specific to a particular application. In either case, these languages are intended to facilitate solving a small range of problems and generally do not have the expressiveness of general purpose languages.

The rule language chosen for *RiverWare* was a general purpose language, although it had some task-specific functionality built into it. The general purpose part of the language came from Tcl, which was its base language. The task-specific parts of the language were subroutines that were added to allow the users to access application-specific information and to provide them with convenience functions. As noted previously, the choice of Tcl was in part due to the fact it was a general purpose language and would provide the flexibility needed to express the users' ill-defined problem descriptions without significant future changes. In this respect, Tcl was a good choice. The flexibility of the language and its expressiveness allowed the users a great deal of latitude in the programming of their rules.

Unfortunately, this same flexibility also made it difficult for other, less experienced, programmers to use the language. In addition, even those with a reasonable amount of programming experience did not have the training or discipline to make effective use of a general purpose programming language. Their programs were often inefficient and difficult to read and maintain. As a result, had time not been the factor it was, a task-specific language would have been the better choice.

4.2.1.2.2. Language Style

A programming language's style is concerned with the presentation of the language. Is the language textual, visual or a hybrid of the two? While this is similar to a language's syntax, it is a broader look at the language and does not delve into the details of specific commands or control structures. It should also be noted that if the programs written in the language are read from or written to a file, an underlying textual representation of the language is required. And even though this textual representation may be accessible to the user, the primary thrust of this discussion is on the application-supported language.

A textual language consists entirely of characters found on a computer keyboard, although it may include syntactic conventions such as using indentation for grouping of statements. This is the style of language with which most programmers are familiar and is the most common form of programming language. A textual language can provide a great deal of expressiveness and flexibility, it can be edited by most any text-based editor and can be easily shared among different users and applications. Another advantage of textual languages is that, assuming the programs are well written, there is generally little ambiguity regarding what programs written in the language mean.

A disadvantage of textual languages is that they generally give the users a great deal of latitude in how their programs are written. While an experienced and disciplined programmer may use the language to write easy to read and maintain code, others may not. In general, textual languages do not put any restrictions on what their programs look like as long as the compiler or interpreter deem the program's syntax to be correct. An extreme example of this is a C program, displayed in Figure 25, that

was entered in *The International Obfuscated C Code Contest* [Phillipps, 1988]. This program, when compiled and run, prints out all twelve verses of the Twelve Days of Christmas.

```
#include <stdio.h>
main(t,_,a)
char *a;
{
return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):
1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/
+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw'
i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n'
')#\
}'+}##(!!/")
  :t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntd-
Ceghiry"),a+1);
}
```

Figure 25. Twelve Days of Christmas in C.

A visual language relies primarily on images and spatial relationships to convey the meaning of the program. While text can and often is included, it is not the sole medium of communication. Text may not even be necessary. Programs written in a visual language are edited using a special editor, although if there is an underlying textual representation, it can often be edited directly using a standard text-based editor. There are benefits associated with visual languages and their use of images and spatial relationships. For example, in *RiverWare*, images are linked together into a network to form a representation of a river basin. These images and links can be manipulated directly to create and update the river basin models. While there is an underlying textual language that can be edited directly, *RiverWare's* visual river basin modeling language and editor provide an elegant means of editing river basins. Another benefit is that the designer of the language can dictate the appearance of the programs written using the language, which can often contribute to the readability and maintainability of the programs written in it.

A disadvantage of visual languages is the ambiguity that can be associated with the programs written using it. Petre stated that while a picture may be worth a thousand words, which thousands words is up to the viewer [Petre, 1995]. For example, in *RiverWare*, the links between objects are not visually directional, even though one object is generally downstream of another. The reason for omitting arrows was that the designers felt that there would be confusion related to the fact that while water flows downstream, information in *RiverWare* can flow in either direction. As a result, the direction that water flows between two objects will not be obvious to a user who is unfamiliar with the river basin.

A hybrid language combines visual and textual language elements. The distinction between this style of language and a visual language is that while there are visual elements, the visual nature of the language does not necessary predominate. An example of a hybrid language is *Mathematica*, which uses special symbols and some spatial relationships to express mathematical expressions. Depending on the problems to be solved, this approach can be superior to the other two in that it can combine the lack of ambiguity associated with text with the expressiveness of special characters and spatial relationships.

I initially wanted the *RiverWare* rule language to be a visual language. This was in part because *RiverWare* used a visual language to represent river basins and I thought a rule language that was similarly visual would be easier for the users to learn and use. It was also in part because I felt that a visual language would be easier for non- to novice programmers to use. Even after the decision was made to use Tcl, I still assumed that either a visual overlay would be added or a completely new visual language would be designed. As will be discussed in the next chapter, upon further reflection, a visual language did not seem to adequately fit the needs of the users and was ultimately rejected.

4.2.1.2.3. Language Paradigm

A number of different programming language paradigms exist. They include procedural, functional, logic-based and object-oriented. I selected a procedural language largely based on the users' preferences. They were familiar with procedural programming languages, needed to translate the *CRSS* rules from FORTRAN into the new language and had explicitly requested a language with procedural

programming features. While the language allowed a subset of users to translate their *CRSS* rules relatively quickly, it did not adequately meet the needs of all its users. Users with little programming experience found it difficult to create, read and maintain these and other rules and more experienced users found it difficult to create readable, maintainable and even correct rules. Accordingly, it can be argued that a procedural programming language was not appropriate for the users of this system.

4.2.1.2.4. Language Features

This last section is a bit of a catchall and contains different language features that were part of the *RiverWare* rule language and created either problems or confusion for the users. All of these features were related to the use of Tcl and included its global variables, pass-by-reference mechanism and general syntactic conventions.

4.2.1.2.4.1. Global Variables

Tcl's global variables are a standard part of the language and serve the expected purpose of providing variables that are global in scope. I was aware of the fact that Tcl included the ability to define global variables and that their use could present both dangers and benefits. Accordingly, I talked to the users about the instances in which I thought the use of these variables might be appropriate and where I thought their use might be inappropriate. Not surprisingly, the users made use of global variables in their rules. Some of these uses were appropriate while others were not. Yet, even the appropriate uses were only appropriate because of deficiencies in the language.

The appropriate use of global variables involved getting the values of object.slot variables at the beginning of a rule's execution. This had two benefits. The first was that it allowed the rules to execute more efficiently since the functions to access these object.slot values only had to be called once. The second was that it allowed the rules to make use of a variable name that was much easier to read than a call to `C_GetValue`. Unfortunately, the manner in which these global variables were used resulted in a loss of at least some of the sought after efficiency gains. One function was used to declare

and get all values used by a number of rules. Since most of these rules only made use of a small subset of the retrieved values, the rules ended up making a large number of unnecessary value access calls.

The inappropriate use of global variables led to a rule that was very difficult to read and maintain. In this case, the user defined a global variable that was written within one function and read within another. These two functions were otherwise independent. This meant that the order in which the functions were called was important, although this was only evident by studying the code. This use of functions with side-effects and the effect it had on the readability and maintainability of the rule did not seem to concern the rule writer. This was mainly due to the fact that the use of global variables made the rule easy to write and produced the correct results. Had these been the main criteria for successfully evaluating the language, one might argue that this was appropriate. Yet, since there were other criteria, including readability and maintainability, this use of global variables was not considered appropriate.

4.2.1.2.4.2. Pass by Reference

Tcl provides an approximation of pass by reference. It involves passing the name of the variable and then having the called subroutine use the “upvar” command to look into the caller’s scope in order to get and use the variable. While this works, it is clumsy and difficult to use. In general, the users made little use of this mechanism. Initially, this was because they did not realize that there was any way to pass parameters by reference. Yet even after they found out about this mechanism, they generally did not want to use it given its awkwardness. As a result, when a subroutine needed to calculate and return more than one value, the users would generally write a function that returned a list of values that would have to be unpacked by the caller. This latter approach was not much better than Tcl’s pass by reference, although the users preferred it.

4.2.1.2.4.3. General Syntax

Tcl’s syntax, as was briefly described in section 3.5.6.1, includes many conventions that differentiate it from more common programming languages. These differences made the rule language more difficult to learn than a language such as FORTRAN. Although the users who made frequent use of the language were able to learn and use it fairly quickly, others who either used the language

infrequently or who only read the rules written in the language found its syntax confusing. This latter group, the readers, were primarily interested in using the rules as a means of understanding the policies that they represented. Yet, since they could not read Tcl, this made it difficult for them to do so. And while they could learn Tcl, it was unlikely given their jobs and priorities that they would do so. As a result, the USBR rule writers needed to translate their rules into a form that could be understood by these users. This need is a clear indication that the rule language did not sufficiently address the needs of either set of users. The readers needed a translation and the writers needed to go through the extra effort of translating.

4.2.2. Subroutine Library

I originally did not consider the subroutine library to be a distinct part of the language. In fact, I did not even think about the explicit need for subroutines. In part this was because I thought of the subroutines as part of the language. It was also in part because I assumed that most of the functionality I would provide would be built into the language. The need for a suite of subroutines came about with the decision to use Tcl as the base language. Tcl provided the basic language syntax, but did not provide any built-in mechanism to support the functionality to tie the language to the application. This functionality had to be provided by predefined subroutines.

Although this need for subroutines was predicated on the use of Tcl, it is likely that any existing language that is used as a base language will require a suite of subroutines to tie it to the application. Even if there are not subroutines in the true sense of the word, the corresponding functionality will be needed. For instance, a language may exist in which access to the state variables within the application is allowed via some mapping between a name in the language and the state variable. Even though a subroutine is not used to access the value, the same issues must be addressed, which may include which values can be accessed for reading and writing, the scale the values are returned in, how multiple values are accessed and the syntax used to access the value. If the mapping is used to get date/time information relating to the application, there will have to be decisions made regarding which date/time components are accessible.

I did not plan in advance the types of subroutines that I would need. I had some ideas about what was required and these included subroutines to read and write the values of object.slots in the *RiverWare* model and subroutines to provide the current simulation date/time so that the rules could make decisions based on when they were executed. The subroutines that eventually became part of the language were added as needed over most of the design and development of the language. These included domain-specific, convenience-related, date/time access and manipulation, and unit conversion subroutines.

Domain-specific subroutines provided water resources- and *RiverWare*-related functionality that simplified the creation and final form of the rules. Convenience-related subroutines allowed the users to simplify their rules by encapsulating multiple actions into a single subroutine. The need for these subroutines became apparent as the users created their rules. Date/time access and manipulation subroutines were added so that the representation of the date/times did not have to be taken into account within the policies. Originally, I had assumed that the rules could directly manipulate the date/times, since they were represented as strings. While this assumption was correct, it had a number of unintended problems. It made the rules less readable, required more work from the rule writers and tied the code in the rules to the representation of the date/times. The latter meant that I could not change the representation of the date/times without requiring the users to update all of their policies. Unit conversion subroutines were provided because the language did not automatically convert values into compatible units. See Section 3.7.1.4 for discussion of the different subroutine types and Appendix E for a complete list of subroutines provided.

4.2.3. Programming Environment

The programming environment provided functionality to access and use the rule language. This included means of loading, viewing, editing and debugging rules. While much of this functionality was not strictly necessary, its exclusion would have seriously undermined the usability of the language. What is notable about the programming environment is that it entailed less planning than the other parts of the language system. This is in part because of the lack of user-specified problem descriptions and in

part because of my lacking the time to design the environment and solicit feedback from the users. This section will describe some of the more interesting aspects of the programming environment.

4.2.3.1. Loading and Saving

The loading of rules into *RiverWare* was necessary to associate rules with the application so that they could be used during simulations. I provided a means of selecting and loading a set of rules and provided feedback on the loading process and outcome. In addition, if the set of rules was not syntactically valid, the user was given a trace and the ability to quickly open the offending file. Section 3.7.2.1 contains a more complete description of loading rules into *RiverWare*.

While the loading of rules into *RiverWare* worked well, the saving of rules did not. In fact, it was not possible to save rules that had been loaded into *RiverWare*. This might have been acceptable had I not provided a means of editing the loaded rules and instead required the users to edit the files that they had loaded and reload them. Yet, as will be discussed in the next section, I did provide some nominal editing capabilities, which when used resulted in inconsistencies between the version of the rules that were currently loaded and those that had originally been loaded. This meant that if they wanted to retain the changes that they had made using the programming environment, they would have to make those same changes to the originally loaded files.

This omission was not intentional. It resulted from a combination of the users' desire to change the priority of the rules using the programming environment rather than having to edit the underlying files and the manner in which the rules were integrated into *RiverWare*. Although it would have been relatively easy to make the changes necessary to support saving of rules, the users did not want this enough to shift my priorities.

4.2.3.2. Viewing and Editing

A number of views of the loaded rules were provided by the programming environment. These included a hierarchical view of the ruleset and all its associated files, a view of the prioritized rules, a view of the rule's dependencies and a view of the rule processor's agenda. These views were primarily

developed to help me debug and test my code as I was developing the system and they eventually turned into pieces of the environment. Unfortunately, they were not entirely useful to the users. In fact, only the view of the prioritized rules was regularly used. This interface was briefly discussed in Section 3.7.2.2 and allowed the users to both see and edit the priorities of the loaded rules.

With the exception of the rule prioritization interface, all editing of rules required the use of an external editor. And, as was noted above, even the use of the rule prioritization interface required the users to eventually edit the underlying rule files. As was discussed previously, this editing required a UNIX-based editor, such as *textedit* or *vi*. While the USBR users liked this and found it convenient, other, less experienced users did not. In general this was based on their unfamiliarity with these UNIX-based editors.

In addition to editing rule files, users with little UNIX experience had trouble creating and maintaining their ruleset and rule files. This became clear during the user test when some of the users were unable to create a ruleset from scratch. The programming environment should not have required the users to use the operating system to perform a task that should have been provided for by the environment itself. Accordingly, a simple and important addition would have been the inclusion of a means of creating a simple, but complete ruleset for the users to use as a starting point. In addition, a means of adding and removing rule files would have been necessary as well.

4.2.3.3. Debugging

The programming environment provided a means of debugging the rules that were written using the language. Although I had discussed the inclusion of some form of debugging with the USBR users, no firm decision had been made as to the nature of this functionality. Options included the use of a Tcl symbolic debugger and the use of user-defined print statements in the rules. Although an augmented version of the latter was chosen, this was not as a result of careful design. This debugging functionality, like the ruleset views discussed above, was originally developed to support me in my testing and debugging of the execution of the rule language. Yet, unlike the ruleset views, this

functionality proved to be very useful to the users. Although this will be treated more fully in a following section, the main reason for this difference was the fact that the debugging functionality supported a user goal, correctness, while the ruleset view functionality did not. Instead, it supported a developer goal, which also happened to be correctness.

4.2.3.4. Miscellaneous Utilities

The USBR users asked for two other features early in the design process, although neither was ultimately given to them. The first was a tool to compare rulesets and the second was the inclusion of version control. The ruleset comparison tool was intended to support the determination of how one ruleset differed from another in order to allow the users to isolate how these differences affected the simulation's results. While this was felt to be a useful tool, the use of Tcl contributed to the tool not being created. The use of Tcl meant that unstructured code would have to be compared and that it was unlikely that anything other than a line-by-line comparison of the differences, as is provided by the UNIX *diff* program, could be given. While this might prove useful, it also could result in a great deal of work for what amounted to a difficult-to-read dump of information. In addition, the lack of structure would also result in some degree of ambiguity regarding what was to be compared. It would not be obvious which rules or subroutines needed to be compared since there were no restrictions on naming or how functionality was grouped. While it would be possible to provide a reasonable comparing tool based on user-provided comparisons, this would entail a great deal of work and require the user to have enough information about the two rulesets to know what to compare. This latter point, to a degree, obviated the need for a comparison tool.

Similarly, version control was dropped as an explicit part of the programming environment because there were a number available through the UNIX file system. These included *RCS* and *CVS*. While it would have been possible to provide a custom tool, even one that relied on UNIX, it was a lower priority than other work items. Yet, given the problems that the users had with using UNIX to create and update rulesets, it is likely that the work involved to do this would have probably been

worthwhile. This can backed up by the fact that the users, even the more sophisticated USBR users, made no use of version control during the development of their rules.

4.2.4. Language Processor

The language processor was the part of the rule language system that was responsible for integrating the rules into *RiverWare*. It provided two main pieces of functionality. One was the ability to load the rules from files into *RiverWare* so that they could be used during a simulation. The other was the ability to execute the rules and to have their execution affect the outcome of the simulation, if the logic dictated this.

4.2.4.1. Language Loader

The language loader had two main pieces. One was in the programming environment and was responsible for the selection of the ruleset file and the display of the results of the load. This was discussed in Section 4.2.3.1. The other, which will be discussed here, was not directly visible to the users and was responsible for the parsing and storing of rules as they were loaded. This included relaying any feedback to the user. This aspect of the design and development was relatively uneventful. The main drawback was the fact that the rules were not stored in such a way that they could be easily edited and then saved by the user. This was partly responsible for the omission of ruleset saving functionality.

4.2.4.2. Language Execution Controller

The language execution controller was responsible for executing the rules written in the language. There were two parts to a rule's execution. One was the interpretation of the code that makes up the rules. This involved executing the Tcl code that was contained in the body of the rules and in the subroutines that were called by the rules. Generally speaking, this worked very well, with the exception of some inefficiencies related to not caching the values of requested variables. The other part, on the other hand, did not work as well. It involved the selection of the rule to be executed. As discussed in Section 3.5.8.3, the decision regarding which rule to execute relied on an internal agenda that was in

turn based on user-specified dependencies. The agenda worked well; dependencies only worked well if the users accurately specified them.

Dependencies were added to the language in order to make the execution of rules as efficient as possible. Recall that dependencies were used to insure that rules were only executed when there was a chance that their execution could change the state of one or more objects in the *RiverWare* model. If dependencies were not part of the language, it was likely that the rules would be executed many more times than necessary. If the rules were very simple, this might not be a problem. However, the rules tended to be very complex and relatively slow. In addition, rule-based simulation was considerably slower than simple simulation, for an otherwise identical *RiverWare* model. This was in part because of the execution of the rules and in part because of the way rule-based simulation solved the objects in the system. Since it could not fully solve the system without rules, it spent a lot of time iterating between the rules and the objects. As a result, objects could be solved and rules executed multiple times for each time step. The increased time necessary to solve models using rule-based simulation made it important to not add any unnecessary rule executions.

As discussed previously, the language only checked to make sure that the rule's dependencies corresponded to actual object.slots in the *RiverWare* model. It did not check to insure that the dependencies the users selected corresponded to the object.slots that were accessed by the rule. This meant that the rule could specify any set of object.slots as dependencies, as long as they existed in the *RiverWare* model. The rule could, of course, contain a complete and correct set of dependencies. But the rule could also contain an incorrect or incomplete set of dependencies. And since the dependencies were used to determine which rules were placed on the agenda when object.slot values changed during the simulation, the use of incorrect dependencies could lead to rules being unnecessarily executed or rules not being executed when they needed to be. If the rules were executed too often, this would lead to efficiency problems, although it would not effect the correctness of the results. If the rules were not executed when needed, the simulation might run more quickly, but it was likely to produce incorrect

results. The only circumstance under which the simulation would both run efficiently and correctly was if the rules contained the complete set of correct dependencies and no others.

Requiring the users to select the rules' dependencies presented a significant safety problem. For unless they were both careful and conscientious, the rules would not be correctly used by the rule-based simulator. When this problem became evident, I discussed it with the users and made clear to them the importance of correctly specifying the dependencies. They agreed that this was important and we decided that we could hold off on a code fix until later and, until then, make sure to use the dependencies as intended.

This did not work out, however. As it turned out, even after being told and, I am convinced, understanding the importance of the dependencies, one of the main USBR users was willing to abandon them in order to get the results that he thought were correct. I realized this when he informed me that he felt the simulator was giving him incorrect results and that he needed my help in looking into the problem. He knew what numbers he was expecting from the simulator and found that he was getting different numbers. In fact, he told me that the rules used to produce the correct result and no longer did. While looking into the problem, he came back and told me that he had solved the problem and that it was not related to the simulator. He told me that he had removed all the dependencies from the rules and was now getting the results he expected.

What had happened was while I was looking through and cleaning up his rules, which I periodically did, I had noticed that he was not using any dependencies in his rules. Knowing that this would not produce correct results, I added them. Yet, the user had previously removed the dependencies in order to get the results he was looking for. He did this knowing that this would make the rules only execute once per time step and also knowing that this was not the way rules were supposed to be used by the rule-based simulation engine. He did this in order to get the results he was expecting, even though, as it turned out, they were incorrect. He was purposefully using the dependencies to manipulate the manner in which the rules executed. He felt that they should only execute once per time step and removed the dependencies to insure this. Thus, he was willing to violate the language's rules in order to

get the answers he wanted. Once we identified this and went through the logic of the simulation, we found that the answers given with the dependencies correctly specified were correct and that the logic of his rules was flawed.

An important lesson here is that the omission of an optional part of the language should not lead to incorrect answers, even if its inclusion leads to significant performance gains. One cannot necessarily count on the users to use the language as intended, even when they know how they should use it. Thus, by not specifying the dependencies, the rules should have been returned to the agenda after every use. This was the manner in which I had originally coded it. Unfortunately, I had changed this to address another bug.

4.3. Reflections on the Design Process

4.3.1. Case Study Participants

The participants in the case study were many of the language's ultimate users. Accordingly, their involvement was invaluable and it is difficult to imagine being able to design and develop the language without their participation. During the design and development, a subset of these users provided two main benefits: they served as models for some of those who would eventually use the language system and they provided the problem descriptions that would be used to design the language. Once the language was substantially complete, these and other users furnished feedback that provided insight into both the final language and the design methodology used. This section will discuss the users and some of their characteristics that either influenced or should have influenced the design. Section 4.3.2 will discuss the problem descriptions.

4.3.1.1. Who, When & How

The initial set of users were the USBR policy writers. Their job was to extract the rules from *CRSS* and express them in the new language. They were the main source of input for both the language and programming environment during the case study. This input proved valuable in that they were able to provide the problem descriptions upon which the language was based. In addition, they helped to

define the suite of subroutines needed to express their rules and used and provided feedback on the system as it was designed, developed and matured. As a result, I spent a great deal of time working with them and was heavily influenced by their needs. This helped to produce a language system with sufficient expressiveness to meet most of their needs.

Unfortunately, it also helped to produce a language system that was too complex for many of the ultimate users. Many of these users were from the Colorado River Modeling User Involvement Group (CRMUIG) and, in general, had less programming experience than the USBR users. While I knew that these users would eventually need to use the language, I did not have any contact with them until very late in the process and, as a result, had little understanding of their needs. Initially, I assumed that their needs were similar to those of the USBR users. This turned out to be a serious mistake. Seeking their input during the design process and even at various points during the development would have helped minimize or eliminate some of the more severe mistakes. At a minimum, this would have included eliminating a number of the programming environment tools that were more useful during development than during actual use and developing an alternative to requiring the direct use of UNIX files and editors. While it would not have changed the use of Tcl, it might well have pointed to the need for predefined rule forms or object.slot access preprocessing notation. The former would provide the ability to create simple rules for setting and constraining object.slot values and the latter would have allowed for a more readable syntax. While each of these would have taken time to design and develop, it is possible that the time saved by not developing and polishing the unnecessary tools would have partially or fully offset this.

As it was, both the language and the environment were too complex and, in places, difficult for them to use. Although expediency dictated the use of Tcl, many of the problems with the language and the programming environment could have been identified and corrected early in the design and development of the language system. In hindsight, I would have requested the participation of as broad a range of users as early as possible in the form of programming walkthroughs and user tests. By doing

this, I would have better understood their needs and been able to determine what elements of the language and environment were appropriate given these needs.

4.3.1.2. Characteristics

In addition to the need to identify and seek the participation of a broad range of users, it is also important to understand the language's users. There are a number of characteristic about programming language users that effect how the language will be used and, as a result, what the characteristics and features of the language should and should not be. During the design and development of the language, I did not take into account this type of information about the users. This led to a number of faulty assumptions which included the notion that the users would not have trouble using what I considered a relatively simple language and that the users would use the language in the manner in which I felt it was intended to be used.

This section will discuss the users' experience and abilities, their preferences, reason for using the language and how they will use the language. While each one of these is important to understand and to take into account during the design, they must be balanced. They will need to be taken in the context of each other and sometimes there may be trade-offs between these areas. For example, the users' preferences may not be entirely compatible with their abilities. This was noticed during the case study in that the users wanted a highly expressive language but had difficulty using it, which led to code that was difficult to read and maintain.

4.3.1.2.1. Experience and Abilities

The users' programming experience and abilities should be taken into consideration when designing the language. Languages suitable for sophisticated users may not be appropriate for inexperienced users. This is in part related to the greater degree of exposure experienced programmers have had to various languages and types of languages. It is also related to the overall ability individuals have as programmers. Even within groups of experienced programmers, there can be a wide range of abilities. Accordingly, an accurate assessment of the users' experience and abilities will help to

determine what type of language is appropriate. For example, it is probable that the more sophisticated the users, the more likely a complex or general purpose language will be appropriate, although sophisticated users do not preclude the use of a simple or task-specific language. On the other hand, users who are new to programming or those who are not accomplished programmers will probably need a language that is less complex and more task-specific.

For the case study, Tcl was chosen as the basis for the rule language. As mentioned in the previous chapter, this choice was largely predicated on the need for a quick solution. Even so, with the exception of its clumsy syntax and execution speed, I thought Tcl would be a good choice. Some of the users had used it in a limited context and I felt that it would provide an appropriate degree of expressiveness to allow them to create their rules. In addition, Tcl was a procedural programming language like FORTRAN and I felt that this would make converting their rules from *CRSS* to Tcl relatively easy. To a degree I was correct, in that the Tcl-based rule language allowed the USBR users to successfully create their rules. What I did not anticipate, however, was that the very fact that Tcl was a procedural programming language would lead to difficulties for many of the other users. Although some could program using the language, enough could not. In addition, as became clear during the case study, even those who could program using it, often had neither the experience nor the discipline to construct programs that where always correct, readable and maintainable. Initially, this surprised me. In hindsight, of course, one can remember the difficulty in mastering what are now very basic programming concepts and the time it has taken to be able to write good code.

An example of one user's misunderstanding of how to use the rule language was noticed during the user test, as mentioned in Section 3.6. One user was creating a simple rule that was supposed to constrain the value of an object.slot in the *RiverWare* model. An example of the user's rule is shown in Figure 26. Here, the rule read the value of Mead's outflow for the current time step and set it to a local variable. This local variable's value was then checked to see if it was greater then 10,000 acre-feet/month. If so, it was reset to 10,000 acre-feet/month. While this rule approximated the user's intent, which was to constraint the value of Mead's outflow, it only updated the value of the local variable. In

order to set the value of the object.slot, a `C_SetValue` call would have to be made as part of the consequent of the conditional. Although I am not entirely sure what the user was thinking, I can presume that the initial `set` command was considered some kind of mapping between the object.slot in the model and the local variable. At any rate, once the problem was explained, the user was able to correct the rule.

```
set mead_outflow [C_GetValue Mead.Outflow 1.0 acre-feet/month]
if {mead_outflow > 10000} {
   set mead_outflow 10000
}
```

Figure 26. Example rule body.

While this example demonstrates a rather fundamental misunderstanding of the language specifically and, perhaps, programming in general, there were other cases in which users were unable to express the logic of their policies using the rule language. They simply did not know how to program and had difficulty with many of the basics, such as assignments, conditionals and loops. Again, this should not be surprising since many of the users were inexperienced programmers and the language was not designed to meet their needs.

A point to mention is that it was not beyond these users to learn to program. Yet, as will be discussed in a following section, it was unclear that they really had the desire or requirement to learn to program. Programming, for many of them, was just a means to an end and, in general, they had little desire to learn to write programs that did anything other than generate the results that they needed as quickly as possible.

4.3.1.2.2. Preferences

User preferences can be used to gain an understanding of the languages, language styles and language features with which the users are most and least comfortable. Their preferences may be a result of some previous experience with particular languages or language features or may simply result from their acceptance of other users' opinions. Either way, their preferences are often a metric for how

well they feel they can use the language to solve their problems and, therefore, how amenable they will be to a proposed language. This language may or may not be the best language for the task.

During the initial discussions with the users, it became clear that they had some strong preferences with regard to different languages and language features. As discussed in Section 2.4.2.1, the users had a very strong dislike for LISP. This resulted in large part from a negative experience they had during the development of *RSS*, although they had little, if any, actual programming experience using LISP. And even though I probably could have convinced them to use LISP, it is unlikely that it would have been the wisest choice. In the first place, LISP is a primarily functional language and requires the use of recursion for anything other than very rudimentary programs. This reliance on recursion was also true for *RSS's* rule language and was a feature that the USBR users had explicitly requested not be required in the new language. Instead, they asked to have it replaced with iterative looping mechanisms. In the second place, I wanted the users to embrace the language and not just use it grudgingly. Had I chosen LISP, I suspect they would have had difficulty using it and, as a result, not fully adopted it. Although they might have eventually gotten used to it, it is difficult to know how long this would have taken and how useful others, with even less programming experience, would have found the language.

The users also had a strong preference for a more traditional procedural programming language. This was largely based on the fact that they were familiar with this type of programming language. Making the new rule language a procedural language made this task easier and, as a result, appealed to them.

4.3.1.2.3. Behavior

In addition to the users' abilities, it is also wise to assess how they will use the language. Programmers will not always use a language as it was intended. Often, this results from the programmers' desire to make their programs produce correct results as quickly as possible. A program's readability or maintainability is either not taken into account or is not considered important. Sometimes,

the programmer realizes the importance of readability and maintainability and assumes s/he will get to it later. In general, this short-term focus probably results from a lack of experience, a lack of discipline or both.

Given that many users will use a programming language in unintended ways, it is probably wise to identify and exclude features that can be misused. Identifying these features, however, can be difficult without being able to look at the programs produced by the users. Unfortunately, there may not be the opportunity to do this until it is too late to easily make the necessary design changes. An alternative is to assume that inexperienced programmers or programmers who only program as a secondary part of their job are likely to misuse the language if given the chance. Accordingly, programming languages for these users should be made as safe as possible.

Given that the bulk of the users of *RiverWare's* rule language were either inexperienced or programming as a secondary part of their job, I would have been wise to take this into account. There were two features that were part of the language that were not used as intended. The first, global variables, were a standard part of Tcl and were left in the rule language because I thought it would prove useful and did not anticipate them being misused. Examples of how this feature was misused are included in Section 4.2.1.2.4.1. The second, rule dependencies, was added to the language to improve the efficiency of the rules' execution. An example of how this feature was misused is included in Section 4.2.4.2.

4.3.1.2.4. Desire to Program

The users' desire to program is also an important consideration when designing a language. It is obvious that they will need to use the language, since that is the reason that it is being developed. What is less obvious is whether they are using it simply to produce the results they need or are using it to write elegant code that produces the results they want. While the definition of elegant code can be debated, the intent of making this distinction is to assess how the language is likely to be used. If the users' primary interest is results, they will be less likely to concentrate on writing code that can be easily

read and maintained by others or even themselves. In addition, they are probably likely to program quick fixes to problems rather than try to find a framework within which to solve the problem. I suspect this is similar to writing prose. One can almost always convey an idea regardless of how tortured the wording. It takes time and effort to produce elegant prose and one has to want to take this time and effort.

The users in this case study were all water resources engineers or managers of water resources engineers. Programming was a secondary task that they used to support their work on modeling water systems. Accordingly, their primary motivation for using the language was to produce results. In fact, in their *CRSS* rules, it would not be uncommon to see a mathematical expression that included a seemingly arbitrary number. When asked why this number was there, I would be told that it was needed to produce correct results. In large part this was because much of the work they do involves modeling physical systems and the models they use are approximations of the actual system. As a result, adding a value to make the model correct, while not particularly elegant, is probably far easier and no less incorrect than overhauling the entire model. The downside comes in terms of readability and maintainability.

This philosophy carried over to a large degree in their use of the rule language. They generally wrote rules that produced the answers that they needed. They rarely spent time making their rules more compact or readable and often produced rules that were either inefficient or difficult to maintain. They generally did produce the correct results, however, and if this were the only purpose of the code produced in the language, this would probably be acceptable. Since the code had to be read and maintained by others, this was not sufficient.

4.3.1.2.5. Reason for Using

The users' reason for using the programming language will influence the choice of language. This is similar to their desire to program. It differs in that it centers around whether or not the users are required to use the language. If the users are required to use the language, there is more latitude in what the language looks like. For if there are features that they do not like or do not initially understand, their

need to use the language will help motivate them to find a way to work through their problems. If their use of the language is optional, however, it is more important that the language be appealing, since they may choose not to use it. The appeal may be that it provides them with the expressiveness they need to solve their problems. It also may be that they are familiar with the language.

4.3.1.2.6. Pattern of Use

The regularity and frequency with which a language is used and how its programs are shared among users should be considered when designing a language. If the language is used regularly and frequently by a single user, a premium can be placed on making the language expressive and easy to write. This is because the user will spend a lot of time with the language and his/her programs, which will help him/her remember what was programmed and why. In addition, frequent use of a language can lead to increased proficiency and the ability to create increasingly elegant and readable code. If, on the other hand, a single user uses the language infrequently or sporadically, the language should be easy to read in order to minimize the time the user must spend becoming reacquainted with any previously written programs. Likewise, if code is regularly shared among multiple users, a premium must be placed on readability and maintainability in order to allow different users to read and update what others have written.

4.3.2. Problem Descriptions

Problems descriptions are examples of problems that the users will need to solve using the language. They provide a valuable resource to a language designer, since they show realistic examples of the problems that the users need the language to address. Yet, there are a number of drawbacks associated with their use, as was demonstrated in the case studies. The first is that these problems may not always be complete or correct. The second is that the users may be unable to provide more than high level descriptions of some of their problems. The last drawback is that problem descriptions are just one piece of information that must be considered when designing an application-specific language system, as will be discussed in Section 4.3.3. This section will discuss the four areas of the application-specific language system and how problem descriptions were used in each of them.

4.3.2.1. Programming Language

I made heavy use of the language-based problem descriptions. They constituted the bulk of the user-defined problem descriptions and ultimately provided a great deal of insight into the types of problems that the users would need to solve using the language. Unfortunately, the initial versions were largely incomplete and incorrect. While they provided a good high level description of the policies that the users would need to express using the language, they fell short in many details. As a result, they could not be used as the basis for a programming language design.

Although one might argue that these high level problem descriptions should suffice as a basis for a language design, I do not believe that this is true. They were sufficiently vague that any design that relied on them ran a considerable risk of not providing some important features. Of course, a general purpose language could be utilized and iterated upon until the users were able to express their policies. This is what I did and, as discussed earlier, it did not result in a language that met the users' needs. Alternatively, a more task-specific language could be designed on the basis of these problem descriptions. While this would likely result in a language that worked for the problem descriptions as they were delivered, it might not result in a language that worked for the problem descriptions once they were complete and correct. This could lead to a language that did not meet the users' needs and would have to be either redesigned or have features added to it as the problem descriptions evolved.

A base language was selected when it became clear that the users could not fully specify their problem descriptions without a working language and simulator and that they needed to match *CRSS's* results using *RiverWare*. This leads to a dilemma. Without complete and correct problem descriptions, any language design based on them is likely to be flawed. Yet, without a working system, users are unable to provide complete and correct problem descriptions. Again, the solution to provide a general purpose programming language, while solving this problem, does not necessarily address the other needs associated with the language. These other needs will be discussed in section 4.3.3.

4.3.2.2. Subroutine Library

The users did not provide any explicit problem descriptions for the subroutine library. This was largely due to the fact that I did not initially consider this to be separate from the language. In addition, it is likely that even if I had, the users would not have been able to provide any problem descriptions that directly related to subroutines. Accordingly, it is safe to assume that the programming language problem descriptions can be used to determine the subroutines that will accompany the language.

4.3.2.3. Programming Environment

The problem descriptions that were provided for the programming environment were not very helpful. This was in part because the users had few concrete examples of problems that they needed to solve. Those they had tended to be very high level. While I was able to incorporate those that they gave me, I also ended up adding other functionality that I had used to test and debug the system while it was being coded. In general, these turned out to be infrequently used. The major exception to this was the tracing facility.

It is probably safe to say that had I designed the environment at a relatively high level and had a walkthrough using it, I would have realized that some of the features that I had added were not as useful as I had originally thought they would be. In addition, features such as the reliance on the UNIX file system and UNIX-based editors would surely have been rethought.

4.3.2.4. Language Processor

As mentioned earlier, there were no problem descriptions associated with the language processor. The language-related problem descriptions, along with a review of expert system literature, identified the needs associated with the rule execution order and the dependencies. I do not think this is unreasonable given that the users are unlikely to be concerned or even aware of the role of the language processor. As a result, they are unlikely to come up with specific problems that will address this component.

4.3.3. Assessment of the Design

As mentioned at the start of this chapter, the language system developed was partially successful. It met an important need of the users in that it allowed them to use *RiverWare* to duplicate *CRSS* functionality. In other respects, however, it failed to meet the users' needs. In order to help understand what aspects of the design process worked and what did not, the remainder of this chapter will provide an assessment of the design of the language and its associated components. The assessment will be made from the perspective of the goals for the language system since they can greatly influence the design and final product. Many of these goals were not known in advance or were not taken into account during the design, which contributed to the system falling short in some important respects.

4.3.3.1. Primary Goal

Although there are many goals associated with the design and development of an application-specific language, the primary goal must be to provide a system that can be used to solve the users' problem. This system may not require a language. For instance, the users' problems may be structured enough that a GUI-based solution is sufficient. In fact, depending on the needs of the users, their abilities and even their probable manner of using a language, a programming language may not be simply unnecessary but actually harmful.

4.3.3.2. Practical Goals

The practical goals I had to contend with helped drive the design and development of the language system. Ideally, this would not be the case and the design of the system could be driven purely by the type of language and environment that most closely matched the users' needs. Unfortunately, it was not possible in this case study to do that and I would guess that, in general, practical goals will need to be understood and taken into account for most application-specific language designs.

4.3.3.2.1. Deadlines

Deadlines will generally be a part of any language design effort and will influence design decisions and the resulting language. This was true with the language created for this case study. The

USBR users' need for a working system that included enough of the language and environment to support the creation of their *CRSS* rules greatly influenced the final system. Their initial deadline was quite aggressive and helped lead to the selection of Tcl as the base language. This decision, in turn, led to a series of decisions that resulted in what became the entire rule language system. While it is possible that a better language system would have been designed had more time been available, the presence of deadlines is often advantageous in that it can help to drive decisions and encourage the removal extraneous language and environment features.

4.3.3.2.2. Budget

The amount of money available to design, develop and deploy a language system is probably finite and often less than ideal. This was certainly the case for the rule language system, which had a fairly limited budget. This manifested itself primarily in the degree to which commercial products could be considered for use as a rule language. The commercial products that I was able to find were generally expensive and required run-time costs. The budget did not support the one-time cost of the software and the USBR did not wish to incur the license fees associated with these products. As a result, any such products could not be considered. Fortunately, there were a number of alternatives, such as Tcl, Python, Perl and CLIPS, which had neither up front nor run-time costs associated with them.

4.3.3.3. Language-based Goals

Language-based goals are those that relate directly to the users' language and environment needs. During the case study, the existence of these goals and how they did or should have influenced the language and environment design became clear. Initially, I had only been interested in the first two goals, expressiveness and facility, since I thought that if I satisfied these two goals, I would have a language system that met the users' needs. While satisfying these goals is necessary to the design and development of a language able to be used to express the users' policies, there are other goals that need to be taken into account. This section will discuss these goals.

4.3.3.3.1. Expressiveness

"A programming system shows adequate expressiveness on a problem if there exists a program of reasonable size within the scope of the system that solves the problem." [Lewis, Rieman and Bell, 1991]. This is a fundamental goal of any programming system and Tcl, as a general purpose scripting language, was very expressive. With the addition of the subroutines necessary to support interaction with *RiverWare*, it was sufficiently expressive to allow the users to represent their policies. Although Tcl was chosen, other languages, such as Perl and Python, would have been good substitutes from an expressiveness point of view. In addition, CLIPS, although not a general purpose language would also have worked. Lastly, a custom language could be designed to be sufficiently expressive.

Using Tcl and the accompanying subroutines, the USBR users were able to express all of the *CRSS* rules. Therefore, it was clearly expressive enough to meet their needs. Unfortunately, this very expressiveness was also one of the language's greatest liabilities. It allowed the users access to features of the language that were not necessarily desirable. In addition, it put no constraints on the form of the rules written. The combination of these made for rules that were difficult to read and maintain.

4.3.3.3.2. Facility

The facility of a language can be defined as the ease with which a programmer is able to express his or her programs. Lewis, et al consider a programming system to show adequate facility "if a user can easily write a program that solves the problem, without the need for extensive problem solving or rarely available background knowledge" [Lewis, Rieman and Bell, 1991]. While a greater degree of facility is likely to be a positive characteristic of a programming system, the reason for the system's facility is an important consideration. The system may have a great deal of facility because it maps closely to the problems the users need to solve or because the system contains features that allow the programmer to solve localized problems at the expense of the readability or maintainability of the whole. The first form of facility is a good thing. The second is not.

An example of this latter definition occurred during the programming of the users' rules. One of the more complex rules – *Reservoir Equalization* – was built over a period of time in an exploratory fashion, since the users were not certain what the rule should look like until they had produced results that matched *CRSS's*. As a result, the rule was not well designed. In one location, when the users found that they needed information that was calculated elsewhere, they declared a global variable to gain access to this value. While this solution easily solved the users' problems, it resulted in a function that contained an obscure side effect that greatly impaired the rule's readability and maintainability.

4.3.3.3.3. Readability

Readability can be defined as the ease with which the code written in a language can be viewed and understood by its intended users. Readability is important for both comprehension and future modifiability [Elshoff and Marcotty, 1982, Jørgensen, 1980]. The intended users may be other programmers or those who wish or need to understand the programs written using the language. This definition depends on quite a few factors, including the language syntax and how well the programs are written. The language syntax can be completely controlled if a custom language is designed. If an existing language is used, the language syntax will be largely, if not completely specified in advance. Yet, even in this latter case, there is likely to be more than a single language from which to choose.

How well the programs are written is another matter. This depends in large part on both the abilities and propensities of the programmer. If they are accomplished programmers who wish to create readable code, they can generally do so regardless of the language. If, on the other hand, they either do not know how or do not wish to create readable code, even the most readable language can be used to create difficult to read programs. Recall the code for the Twelve Days of Christmas in Figure 25, which could be written much more clearly with very little effort. In fact, it is difficult to imagine a less readable version of that program.

Readability was an important goal for the users of the *RiverWare* rule language. In the first place, they needed to share the rules with each other. More importantly, however, they needed to show

the rules to other users who were not familiar with Tcl and who often lacked programming experience. As was discussed earlier, they did not feel that the rules as expressed in Tcl were in a suitable form for these users. Since these users would need to view and understand the policies expressed in these rules, it became obvious that the Tcl-based language did not adequately address readability. Although the USBR users could translate the language to a readable form when they needed to show others the policies, this would be time consuming and error prone. Alternatively, those who needed to read the policies could learn Tcl, but since they were not programmers, this might be asking too much.

A last consideration regarding Tcl and, to a degree, any general purpose programming language, was that it provided so much flexibility that it was easy to create very difficult to read programs, its syntax notwithstanding. Using Tcl, the USBR users created programs that were very difficult for even trained programmers to read. While other programming languages, such as Perl, could be used to create even more difficult to read programs, a programming language does not necessarily have to allow this. While a programming language may be designed that does not have the power and flexibility of a general purpose programming language, it can be made sufficiently expressive to be used to solve the users' problems without necessarily attempting to solve any problem that might exist.

4.3.3.3.4. Writability

Writability can be defined as the ease with which a programmer is able to express his or her problems using the programming system. It is concerned with the mechanics of producing the programs that are possible using the programming system. While it is similar to both expressiveness and facility, it does not really deal with either what can be said using the system or how well the programming system provides the means for expressing problems. Some examples might help illustrate how writability differs from both expressiveness and facility. The first involves Perl, which is considered a very writable language because it contains many ways to express the same concept. See Figures 12-16 in Chapter 3 for examples of this. Note that some of these examples require very little writing and, for those who are familiar with the syntax, are very easy to write. Yet, since each of these methods express the same functionality, they do not increase the expressiveness of the language. The second is a more generic

example that involves complex equations, which can be written in infix, prefix or postfix. All show equal facility in that they do not require “extensive problem solving or rarely available background knowledge.” Yet, since most programmers are more familiar with infix notation, this syntax will be more writable for most.

In addition to purely textual considerations, writability can be influenced by the programming system’s environment. An editor can significantly increase or decrease the writability of a language, regardless of the underlying textual language. For instance, Smalltalk contains a large body of classes that can be used to solve a wide range of problems. Yet, the existence of the classes is of little use if the programmers are not aware of them. That is why the main commercial implementations of Smalltalk contain extensive programming environments to aid the programmer in the identification and use of these classes and their methods.

The *RiverWare* rule system provided a modest amount of writability. While it did not unduly restrict the users in the creation and modification of their programs, it also did not provide much assistance. There were a number of factors that contributed to this. Some were practical and to a degree unavoidable. Others were oversights that, with better planning, could have been avoided. The practical factors included the GUI library that was used to code *RiverWare* and the use of Tcl. The GUI library, while quite powerful in many respects, did not provide a text editor with the functionality necessary to support the rule writers. Although I could have augmented the library’s simple text editor to include this functionality, the time and effort involved was too much to justify given the time constraints. And besides, UNIX contained a number of editors that could be easily used. The use of Tcl also negatively influenced the writability of the programming system. This was mainly due to its awkward syntax and the fact that there were a large number of *RiverWare*-specific Tcl subroutines that could be used. The existence of these subroutines was partially mitigated by the inclusion of a dialog that contained a listing of the names and arguments for all of these subroutines.

4.3.3.3.5. Maintainability

Maintainability is the ease with which an existing program can be understood and modified within the programming system. It contains elements of both readability and writability, although it goes beyond these. For example, consider a set of rules that reference a particular *RiverWare* object.slot. If the name of the object.slot is changed in the *RiverWare* model, it must also be changed for each rule in this set. A programming system that emphasizes maintainability might provide a means of making this change globally or at least provide a means of identifying all the occurrences of the object.slot. A programming system that does not emphasize maintainability may require the programmer to manually identify and change each occurrence.

The *RiverWare* rule system provided very little support for maintainability. Most of the support for maintainability was to be found in the UNIX programming tools. While these were quite powerful, they were also cryptic and essentially unusable by the majority of the rule system.

4.3.3.3.6. Familiarity

Familiarity defines how well the programming system maps to the programmers' experience with and knowledge of similar systems. Generally speaking, the more familiar a programmer is with a system, the easier it will be to use, at least initially. Of course, an unfamiliar system may be inherently better, in which case the initial difficulties will probably give way to significantly greater productivity once the system is learned. Another factor regarding familiarity is that is does not necessarily imply anything about the users' preferences. The user may be very familiar with the programming system, but not like it.

The USBR users were familiar with Tcl before the decision to was made to use it as the basis for the rule language. Tcl had been used for the *RiverWare* modeling language and because of various and sometimes temporary limitations in the *RiverWare* interface, the users were often required to edit the model files directly. While they did not become accomplished Tcl programmers by editing the model language, they did gain enough familiarity with its syntax to be able to read and update simple

programs written using it. In addition, since Tcl is a procedural language, it bore enough similarities to FORTRAN that the users could easily understand it control structures. Lastly, they generally liked Tcl, which was probably based in large part on the fact that they knew it and were able to use it effectively.

This familiarity played a significant role in the selection of Tcl as the base language for the *RiverWare* rule system. There were a couple of other alternatives, Python and CLIPS, which might have been better options. Yet, these were rejected in part because of the users' were not familiar with them. One of them, Python, was a syntactically cleaner language and considerable faster than Tcl. The other, CLIPS, was a rule system that could have been easily embedded into *RiverWare*. Although other factors were involved in the decision, these other languages would have received greater consideration had the users not been familiar with Tcl.

4.3.3.3.7. Safety

A programming system's safety can be measured by the number of programming errors that can be caught before they are executed. Errors can be caught at two main times: at load or compile time and at run time. Since the *RiverWare* rule language was interpreted, this discussion will pertain to load time and run-time errors. For simplicity of discussion, this section will discuss safety from two aspects. One will be called security and it relates to the ability of a programming system to reject syntactically invalid programs and to prohibit the execution of illegal actions. Examples of illegal actions include division by zero and writing to a read-only object.slot. The other will be called correctness and it relates to the ability of a programming system to prohibit the creation and execution of programs that contain statements that will lead to incorrect results.

4.3.3.3.7.1. Security

By and large, the *RiverWare* rule programming system was secure. It only allowed syntactically correct rulesets to be loaded. Syntactically invalid rulesets were caught at load time, the loading terminated and the user given a back trace that detailed the error. In addition, it did not allow programs written in it to effect parts of *RiverWare* to which it was not supposed to have access. At run-

time, if an error occurred, it was caught, the simulation terminated and the user notified without any harm being done to the state of any *RiverWare* entities. In this case, too, the user was given a back trace detailing the error and its location within the rule.

4.3.3.3.7.2. Correctness

The *RiverWare* rule programming system had both good and bad aspects related to correctness. There were a number of areas in which the users were able to create rules that were incorrect. Two of them have been discussed previously and were particularly problematic. The first was the use of dependencies. As discussed earlier, the specification of dependencies was optional, yet if the programmer omitted any dependencies, the corresponding rule could not be guaranteed to be correct. This language shortcoming was further exacerbated by one of the users who would not use dependencies as was required. In addition, even if the users chose to use the dependencies as required, it was often difficult to determine what the dependencies should be.

The second was the language's lack of support for insuring the correct usage of values that could have different and sometime incompatible units. Although routines were provided for converting values to different units, the language did not catch expressions that combined values with different or incompatible units. For example, it was possible to compare a flow in cubic-feet per second (*cfs*) to a flow in acre-feet/month (*af/m*) or even to a length. This made it possible to create rules that would often look correct, but would contain difficult to identify errors.

On the positive side, the programming system did provide support for helping the users determine the correctness of their rules. Rule tracing, which amounted to the inclusion of tunable system- and user-specified print statements, proved to be an immense help to the users in their process of insuring their rules were correct. In addition to rule tracing, the ability to set breakpoints and examine the state of *RiverWare* before and after the execution of a rule was also valuable. While not used as frequently as rule tracing, it nonetheless provided a means for the users to insure that their programs were correct.

4.3.3.3.8. Performance

Performance is concerned with insuring that the programs written using the programming system run fast enough to meet the users' needs. For *RiverWare*, the need to execute the rules quickly was predicated on the fact that the users had to run simulations for very large models over many years. The language's performance was considered to be on the slow side, although within the realm of acceptability. The slowness of execution was primarily based on the speed of the entire rule-based simulation, Tcl and how I had written the code. The rule-based simulation was slow because of the manner in which it interacted with the rules. This interaction resulted in an increase in the number mass balances performed during a simulation run. Tcl was slow in part because it was interpreted and in part because of its use of strings for representing data. The code I had written contained some inefficiencies, which included the fact that I neglected to cache variables that the rules read from *RiverWare* objects.

4.3.3.3.9. Deployment Environment

One of the requirements that the USBR had for any rule language system was that it be able to run across all the standard platforms. This included, the dominant UNIX flavors (i.e., Solaris, AIX and HPUX), the windows environments (i.e., Windows NT and Windows 95) and Macintosh. This requirement was to insure that *RiverWare* could be ported to another platform. As a result, I needed to make sure that the options I considered could in fact be used without modification across multiple platforms. Fortunately, all the options mentioned in the previous section met this requirement.

Chapter 5

Case Study 2

"Plan to Throw One Away"

Frederick P. Brooks, Jr.
The Mythical Man-Month
[Brooks, 1982]

5.1. Introduction

At about the time that the USBR had completed and verified the conversion of its *CRSS* rules to the new *RiverWare* rule language system, a decision needed to be made regarding the direction of the project. Should the current language system be refined and improved in order to address the problems that had been identified or should a new language be designed and developed? As discussed earlier, there were both positives and negatives associated with the language system. In the end, the negatives outweighed the positives and a decision was made to develop a new language. This chapter will discuss the issues that surrounded the design of this new language. In order to make the discussion as clear as possible, this chapter is not chronological. Instead it discusses the design context, design goals and the problem descriptions. It will then discuss the language and environment that resulted and discuss the issues that drove the decisions made. Lastly, it will present an overview of the user test on the resulting language.

5.2. Design Context

The design context for this case study was fairly similar to the design context for the first case study. *RiverWare* was largely unchanged and the primary USBR users were still closely involved. In addition, the CRMUIG users were still involved, although as before, their involvement was limited. And while the information I had gained from the first case study had strongly indicated that the language system needed to be changed, I had to contend with an unexpected situation. Some of the primary

USBR users had grown attached to the initial language system and were hesitant to change, despite their knowledge of the problems associated with it. As a result, I had to carefully consider how I would make the necessary changes to the new language system without unduly alienating those users.

The main differences in the design context centered around the problem descriptions and my increased understanding of the design goals for the language. The problem descriptions were the same in concept, but were now much more fully specified in that they had been made into complete rules in the original system. These more fully developed problem descriptions provided a clear picture of the type of problems that the users would need the new language system to be able to address. The design goals for the language helped to provide the context within which these problem descriptions would be used. They helped to narrow the choice of possible languages by focusing the language design on both the needs and characteristics of the users and the problems that they needed to be able to represent.

5.3. Design Direction

The first design decision that had to be made was whether the current language system should continue to be used or a new language system designed and developed. This section will discuss the different options that were considered and the final selection.

5.3.1. Refinement

The first option was to stay with the original programming system and clean up many of the problems associated with the language and programming environment. This option would be the easiest and quickest to implement and could be used to address many of the ills associated with the initial system. In addition, as stated above, the USBR users had grown somewhat attached to the system. This option would have required the continued use of Tcl for the creation and modification of the users' rules. As a result, it would not adequately address some of the main problems associated with the language system, which included Tcl's awkward syntax, its susceptibility to misuse and its poor performance. These considerations led to the rejection of improving and extending the original system.

5.3.2. Visual Overlay

Another option considered was the use of a visual overlay. This option was first envisioned as the probable next step in the evolution of the rule language system. In fact, the use of Tcl as the language with which the users directly interacted had been considered a short-term solution from the start. Yet, during the design and development of the original language, it became clear that a visual language would not be as useful as a textual one. This was mainly due to the fact that the users' problem descriptions needed to be represented unambiguously in order to insure that the policies were easily understood by others. Given that the policies consisted primarily of numerical and logical expressions, it was felt that the programs written using a textual language would be less prone to misinterpretation than programs written in a visual language.

This conclusion was supported by the visual language literature, which made it clear that visual languages are not necessarily more readable than textual languages [Petre, 1995, Green and Petre, 1992, Petre and Green, 1990, Petre and Price, 1990]. One of the primary reasons for this is that the pictures used to represent programs in visual languages are often open to interpretation. While a picture might be worth a thousand words, it may not convey the same thousand words to all viewers [Petre, 1995]. Given the need for unambiguous rules and the fact that textual alternatives existed that could be used to represent rules in a manner that was familiar to the intended users, the creation of a visual overlay was rejected.

5.3.3. Textual Overlay

The use of textual overlay, on the other hand, had some potential. A textual language could be designed that would smooth over the syntactic difficulties of Tcl and the language's subroutine library. It could minimize the lack of structure inherent in Tcl and thus provide a considerably more readable and writable language. It would require a preprocessor that would convert the users' rules into Tcl. The main negative associated with a textual overlay was the use of Tcl. Since Tcl was perceived to be slow, a textual overlay would be at least as slow as native Tcl. And even though the overlay language could be converted prior to run-time, it was possible that the translation could introduce inefficiencies. Yet even if

it did not, the speed of Tcl at its fastest was considered too slow. As a result, this option was rejected as well.

A variation on the use of a textual overlay was to use a faster language as the base language. Possible base languages included Python and Perl. Whereas these languages are considerably faster than Tcl, their use would require both the design and development of a new base rule language and the creation of an overlay language. In addition, both of these languages were general purpose and procedural and would not address the associated problems that the users had had with Tcl. This and the risk that use of these languages might not provide the flexibility needed to include features such as unit checking and automatic unit conversion, led to the decision to reject this option.

5.3.4. New Language

The last option was to design a new language from scratch. This would allow the greatest degree of flexibility in meeting the users' goals. It would also entail a considerable amount of work. Yet, given the perceived problems associated with the other approaches, this option was felt to be the best approach.

5.4. Design Goals

The design goals were integral to the design of the new language system. They helped make clear what decisions should be made as the design of the language progressed. As was discussed in Section 4.3.3, the design goals for the new language system were more clearly understood by the time the second case study was started. This was in large part due to my observations of the USBR users as they worked on their rules for the first language system and examinations of these rules as they were being written and once they were completed. It was further confirmed by the user test with the CRMUIG users.

Early in the design of the new language, the design goals did not map to specific solutions. As the design progressed, however, they had a great deal of influence on the decisions made. While each of

the design goals were relevant to the design of the second language, readability, writability, safety and performance were the primary goals since they addressed significant deficiencies in the first language.

5.5. Problem Descriptions

In order to determine what the new language should look like, I used the rules written by the USBR users. These rules were fundamentally complete and correct and, given the amount of time I had spent examining their programs, I had a high-level understanding of both the underlying policies and how they needed to interact. My knowledge of many of the rules' details, however, was shallow.

While these rules generally represented the policies and even produced the correct results, they had some important drawbacks. They contained errors in code not executed for the *CRSS* results test and misused features of the language. In addition, they often contained inconsistent indenting and spacing, which, while easy to rectify, impeded readability. Lastly, the rules made use of functions with side-effects, which resulted in rules that were easy to break while making seemingly innocuous changes.

5.5.1. Simplification

Since I had decided to design a custom language, I needed to fully understand their rules. To this end, I had decided to work through their rules in order to clean them up and disambiguate them. I started by selecting one of the simpler rules: *Blue Mesa Rule Curve*. An example of this rule, written in the pseudo-code used to disambiguate the users' rules is found in Appendix C. All of the functions and the portion of the rule that sets `blue_mesa_data.target_storage[]` were used in the final rule.

I started at the topmost level of the rule and began to drill down to the details. This worked for a while, although it became clear that I needed to understand the concepts upon which the rule was based before I could understand the logic at the higher levels. This entailed defining the lowest level concepts and building upon them to form more complex concepts. The concepts were essentially functions that were generally arithmetic expressions of object.slot values, literal values and other functions. A number of these functions also needed to include conditionals.

In the beginning, I defined the functions without arguments; they consisted of a name and a definition. This was mainly an attempt to keep them simple, even though I was confident that I would have to include arguments at some point. With or without arguments, the functions had the benefit that they had no side-effects. As a result, they could be examined and understood without evaluation. In addition, these functions could be arbitrarily combined without concern for the order of their placement. While I suspected I was heading toward a functional language, I tried not to bias my design until it became clear that this was the best solution.

5.5.2. Building

As I built up the rule using these functions, it became apparent that there needed to be a way to set the value of an object.slot. There were two obvious approaches. One was to use a predefined function that took the name of the object.slot to be set and the value to which it was to be set. This was the approach taken in the first language and, while it would provide the necessary functionality, it did not provide a particularly readable solution. The other alternative was to provide an assignment statement as part of the language. An example of such a statement is `object.slot[] := expression`. This statement would assign the value associated with `expression` to `object.slot`, where `object.slot` could be qualified to include the date and time, within the square brackets, at which the value would be set and `expression` would be an arithmetic or logical expression, that was based on the values of object.slots, literals and functions. While this did not fit exactly into what was a purely functional language, it did seem to provide the needed functionality in a readable form.

The permissible locations of assignment statements also needed to be determined. One option was to allow them inside a function, which would permit functions to set values during their execution. This would have two negatives impacts. It would complicate the language from both a design and a readability perspective. From a design perspective, it would require a function to compute a value, set it to an object.slot and then return it. This seemed clumsy at best, since it required the creation of functions with side-effects. From a readability perspective, it would allow the setting of values deep within the logic of a rule, which was also allowed in the previous language. This feature required that all

the code in a rule be inspected in order to know which object.slot values might be set by the rule. Given these problems, it seemed best to allow assignment statements only at the highest level of the rules and outside of functions. Thus, a rule at its highest level would consist of one or more assignment statements. This would allow a reader of the rule to immediately know which object.slot values the rule might set. For more details about assignment statements, see Section 5.6.1.4.

At this point, I had established a number of the basic components of the new language, which consisted of the function, the expression and the assignment statement. Although they were still somewhat ill-formed, I had been able to use them to write the *Blue Mesa Rule Curve* rule in a form that I considered much more readable than the original language. Yet, I wanted to make sure that both the USBR and the CRUIMG users understood this embryonic new language and felt that they could understand the rules written in it and use it to create their own rules. I initially sought feedback from the USBR users. I showed them some of their rules represented in this new language. When I received positive feedback, I took these same rules to the CRUIMG and solicited their feedback. When they also gave me positive feedback, I started the process of refining the language design.

5.5.3. Value Constraints

Once I had the users' approval, I sought to expand the new language to include the remaining parts of the USBR's rules. One interesting and frequently repeated pattern was the computation of a value followed by logic to constrain it in order to insure that the value did not fall outside an acceptable range. For example, to prohibit the illegal addition of water into the system some rules computed and then constrained evaporation to a value greater than or equal to zero. A similar pattern involved a range check, but instead of resetting the value if it was out of range, it contained an instruction to abort the simulation. This type of check was used to catch instances in which the computations within a rule were so unacceptable that there was no point in continuing the simulation's execution. Since both of these types of value constraints were common, it was necessary to insure that the language could represent them.

While these value constraints could be explicitly programmed by the user, it would impair both the function's readability and writability. For example, consider Figures 27 and 28. In Figure 27, the function writer is responsible for constraining the value. In Figure 28, the function itself is responsible for constraining the value; the function writer is only responsible for specifying the constraint. Both functions accomplish the same task, although the function in Figure 28 is more concise and easier to read. And only one line per constraint needs to be added to the function in Figure 28, while up to five need to be added to the function in Figure 27. Although other possible representations could be used in the function in Figure 27, they are unlikely to be as straightforward as the notation in Figure 28.

```
FUNCTION ComputeMeadEvaporation ()
{
   mead_evaporation := ...
   if (mead_evaporation < 0) then
      return (0)
   else
      return (mead_evaporation)
   endif
}
```

Figure 27. User-defined value constraint.

```
FUNCTION ComputeMeadEvaporation ()
   MIN_CONSTRAINT: 0
{
   mead_evaporation := ...
   return (mead_evaporation)
}
```

Figure 28. Declarational value constraint.

5.5.4. Types

The language needed to support different language types to allow the users to represent their rules. The main type needed was numeric, since *RiverWare* used them in all of its calculations and the rules would need to read, write and operate upon them. There was not a compelling reason to distinguish between integers and real numbers, since *RiverWare* did not distinguish between them and I wanted to keep the language as simple as possible. As a result, all numbers were real numbers by default.

In addition to single numeric values, lists of numeric values were needed, since the original rules would often get and manipulate lists of object.slot values. Some of these lists needed to be summed or averaged and others needed to be iterated upon. The last type needed at this point was boolean, as this type was used in the antecedent of conditional expressions. Since object.slots could not hold boolean values, the main source of these boolean values would have to be logical or relational expressions. As it turned out, these types were just the beginning. See Section 5.6.1.6 for a discussion of all the types used in the language.

5.5.5. Open Issue

At this point, it was unclear how to handle iteration and/or recursion. Given that I had selected, albeit unintentionally, a primarily functional language, the natural choice was recursion. Yet, the users had explicitly requested a language without recursion, which made for a potentially difficult decision. A number of factors, however, convinced me that I should defer this decision until later. The main one was that the design of the language thus far had already addressed a number of serious problems associated with the original language and I felt it important to build upon this foundation. The other was that I felt that control structures could be developed that would satisfy the users without resorting to recursion. For instance, I could provide overlays that would mask the use of recursion or, as a last resort, I could persuade the users that the benefits associated with the new language would outweigh the difficulties associated with recursion.

5.6. Language

5.6.1. Elements

This section will discuss the language and its various elements and give motivations for their inclusion in the language. The elements include both high-level structures and the underlying foundation of the language.

5.6.1.1. Rulesets

The ruleset was part of the original language and consisted of a single file containing names of files containing rules. These files, called rule files, could include references to files that contained functions. While this provided the users with a great deal of flexibility, it resulted in a large number of files that had to be created and maintained by the users. This reliance on files coupled with the lack of programming environment support presented several problems. The first was that it required users who might not be familiar with the file system to learn enough about the file system and associated utility programs to properly create and manipulate their rulesets. As was demonstrated during the initial user test, this proved to be difficult for some users. The second was that it required users to manage a large number of files that together represented a single set of rules.

There were at least two solutions to this problem. One was to leave the ruleset file with its associated rule and function files as is and augment the programming environment to shield the users from the file system. The other was to place the entire ruleset in a single file and provide programming environment support to allow for manipulation of the ruleset and its individual parts. While both options would shield the users from the file system, the latter option was preferable since one file could more easily be manipulated than a number of files. The main disadvantage of this approach was the loss of flexibility. For example, since a rule would no longer exist in its own file, the user would have a more difficult time saving or copying a single rule. As will be discussed later, this liability could be and was addressed with functionality provided by the language's programming environment.

5.6.1.2. Groups

For organizational purposes, the USBR users stored their rules in various directories, grouped by the overall policy that these rules represented. For instance, all rules associated with the *Mead Flood Control* policy were placed in a subdirectory called MeadFloodControl. In addition to the rules, the subdirectories also contained functions that were specific to the subdirectory's rules. Functions that were not specific to a particular policy's rule, but were shared by different policy's rules, were stored in separate, common directories.

The manner in which the USBR users' organized their rule and function files motivated the creation of groups within a ruleset. And given that the users had made use of two different kinds of subdirectories, those for rules and functions that were part of a larger policy and those for functions that were shared by multiple policies, I decided to create analogous groups called policy and utility groups. Policy groups would be sets of rules and functions that were meant to form single, coherent policies. They could contain any number of rules and functions. The rules would be global in the sense that they were part of the ruleset and needed to be executed in the context of all other rules in the ruleset, regardless of the policy group to which they belonged. The functions stored in a policy group would not be global and could only be used by the rules within their group. Utility groups would be sets of functions that were intended to be used by any of the rules or functions in the ruleset, regardless of the group to which they belonged. Accordingly, these functions would be global.

5.6.1.3. Rules

As with the initial language, rules were needed in the new language. Yet, given the more structured nature of the language, the new rules had some fundamental differences. Perhaps the most obvious was the use of assignment statements as top-level constructs. And in order to provide the expressiveness necessary to allow the users to create their policies, a number of variations on the assignment statement were included. These will be discussed in Section 5.6.1.4, which discusses rule statements. In addition, since rules in the previous language often set multiple values, multiple assignment statements were allowed.

An additional feature gleaned from the USBR's rules was the use of a conditional at the start of the rule that served as a means of determining whether or not the rule should be executed. For instance, the *Surplus* rule was only allowed to execute in January. This type of execution constraint was similar to the function's post-evaluation constraints discussed earlier. Therefore, I made the decision to include a declarational pre-execution constraint as part of the rule. Like the function's post-evaluation constraint, this would not be part of the main body of the rule, but would be a separate expression. It would be

executed by the language processor before the rule was executed. If this expression evaluated to true, the rule would be executed; if it evaluated to false, the rule would not be executed.

As with the function's post-evaluation constraints, the rule's pre-execution constraint was not strictly necessary. The rule writer could include this condition as part of the rule's assignment statements. Yet a number of factors convinced me that keeping these rule-specific conditions separate was the better approach. The first was the frequency with which the users made use of this type of constraint. The second was the fact that it made the conditions under which a rule could be executed explicit and separated these conditions from those that might dictate whether a particular object.slot should be set. This latter issue will be discussed in the following section. An example of the pre-execution constraint is in Figure 29.

```
RULE Surplus
   EXEC_CONSTRAINT: MONTH == JANUARY
{
   ...
}
```

Figure 29. Pre-execution constraint.

5.6.1.4. Rule Statements

The statements that the rules would use to compute values and assign them to object.slots also needed more complete specification. Initially, only one type of statement, the assignment statement, was included. Yet, it soon became clear that others would be necessary. These included conditional assignment statements and print statements. The following sections will discuss the different types of rule statements and the issues associated with them.

5.6.1.4.1. Assignment Statements

As discussed previously, the rules needed assignment statements to set values of object.slots in the *RiverWare* model. The initial conception of these statements, `object.slot[] := expression`, where `expression` could be an arbitrarily complex arithmetic expression, remained fairly consistent.

As the next section will discuss, however, some changes had to be made to accommodate instances where the assignment was not always desired.

5.6.1.4.2. Conditional Assignment Statements

Conditional assignment statements were a means of allowing the rule writer to assign a value to an object.slot only if certain conditions were met. For example, consider a rule that restricts the flow of water out of a reservoir to its maximum allowable flow. If the flow is above the maximum, it must be reset to the maximum. If the flow is less than or equal to the maximum, nothing should be done. Although this type of condition is similar to the pre-execution constraint on a rule, it only applies to a single assignment and each assignment could have its own conditional. The rule's pre-execution constraint, on the other hand, applied a single constraint to the entire rule.

While it was clear that a conditional rule statement was needed, the appropriate syntax was less clear. There were two main options. One was to use the traditional procedural notation, as in Figure 30. This had a number of appealing aspects, which included a familiar notation and the ability to include multiple assignment statements within a single conditional. It had the disadvantage that it would lead to arbitrarily deep nesting of conditionals, which could serve to obscure the values being set and the conditions under which they were set.

```
IF (month == January) THEN
   mead.outflow[] := 5
ENDIF
```

Figure 30. Procedural conditional notation.

Another option was to use a less traditional approach, as in Figure 31. This approach had the advantage that it made it clear what value was being set. It also had the advantage that the form of the assignment statement could remain `object.slot[] := expression`, where `expression` could be expanded to be an arbitrarily complex expression that could include a conditional. It had disadvantages that included unusual syntax and the fact that if the user wished to set multiple values based on the same condition, the conditional would have to be repeated. This could be partially mitigated by using a

common function in the antecedent of the conditionals. Despite these disadvantages, the advantage of making clear what values could be set by a rule led to the selection of the option in Figure 31.

```
mead.outflow[] := IF (month == January) THEN
                     5
                  ENDIF
```

Figure 31. Non-procedural conditional notation.

5.6.1.4.3. Print Statements

In addition to setting values in the *RiverWare* model, the users also needed a means of displaying the status of the execution of their rules for debugging. The initial language had provided this capability and the users had made abundant use of it. Yet, given the manner in which the assignment statement was constructed, with one expression per statement, an embedded, free-form print statement was not appropriate. An alternative was to allow for an explicit print statement of the form `PRINT expression`, where `PRINT` was a pre-defined keyword and `expression` was an arbitrarily complex and possibly conditional string expression. While this approach was consistent with the assignment and conditional statements, it suffered from the same problem as the conditional statement in that users might need to duplicate conditionals.

5.6.1.4.4. Assignment Order

An issue related to the assignment of values was how multiple assignments that might rely on each other should be handled. Should the result of one rule's assignment statements be available to one of its subsequent assignment statements? As an illustration, consider the example in Figure 32, which contains the initial values for two object.slots and two assignment statements that will be made on these object.slots. The initial value for `mead.outflow` is 5 and the initial value for `powell.outflow` is 6. If the result of the first assignment is available to the second assignment, both `mead.outflow` and `powell.outflow` will have final values of 0. While this seems reasonable, it implies that the order of the assignment statements is relevant. Switching the order of the two assignment statements will result in `mead.outflow` getting the value 0 and `powell.outflow` getting the value 5. While the dependency

is obvious in this example, it is easy to imagine more complex examples in which reordering of assignment statements would cause unexpected results. Although this type of dependency made sense for a procedural language, it meant that rules could not necessarily be understood outside of the context of their execution.

```
Initial values:
   mead.outflow[] = 5
   powell.outflow[] = 6

Assignments:
   mead.outflow[] := 0
   powell.outflow[] := mead.outflow[]
```

Figure 32. Simple, multiple assignment rule.

An alternative solution was to consider the assignment statements a system of equations that needed to be satisfied. If there was a change to one or more dependencies, the equations would be solved using the new information until they were all satisfied and there was no new information. Thus, given the switched ordering of assignment statements from Figure 32, setting `mead.outflow` would result in the re-evaluation of the `powell.outflow` assignment, which would in turn result in `powell.outflow` being set to 0. This solution would solve the problem associated with the relevance of execution order, but would be relatively complex to implement. It could also lead to circular definitions. For example, consider the two assignment statements in Figure 33, in which both the values for `mead.outflow` and `powell.outflow` are continually incremented by one.

```
mead.outflow[] := powell.outflow[] + 1
powell.outflow[] := mead.outflow[]
```

Figure 33. Assignment statements that would lead to an infinite loop.

At least one other solution existed that was far simpler. This solution was to take a snapshot of the state of *RiverWare* before the rule's execution and to the use those object.slot values throughout the rule's execution. Thus, the assignment statements would not be taken into account until after the rule completed its execution. In the example from Figure 32, `mead.outflow` would be set to 0 and `powell.outflow` would be set to 5. This turned out to be the solution that the USBR selected.

5.6.1.4.5. Constraints

Another consideration in the design was to include post-evaluation constraints on the assignment statements like those used with functions. While this would have been consistent in some respects, it would have led to notational problems. It would be difficult to show these constraints within the context of the rule without seriously impairing the readability of the rule and each of its assignment statements. It would not necessarily add a great deal in terms of functionality, since the right-hand side of the rule's assignment statement could be a function, which in turn could have post-evaluation constraints. Thus, even though it might have added some additional functionality and flexibility, the costs did not outweigh the benefits. As a result, I decided to keep rule assignment statements as simple as possible.

5.6.1.5. Functions

As already discussed, functions were an important part of the new language. In fact, given that the language had turned out to be primarily functional, they played a central role. Although I had originally intended to provide just one type of function in the new language, this changed as the design progressed. The following sections will discuss the two types of functions that were included in the language.

5.6.1.5.1. Internal Functions

Early in the design, I had envisioned one type of function that consisted of a single expression that evaluated to a value or a list of values. This type of function came to be known as an internal function. While the function's expression could be as complex as the user desired, there could only be one. This made the internal function considerably simpler than the functions that were allowed in the original, procedural language and had few restrictions with regard to their content. My main motivation for the heavy restrictions on the form of the internal functions was to help insure that the functions and the concepts that they represented were kept as clear as possible.

5.6.1.5.2. External Functions

As the design progressed, however, I came to the conclusion that I needed an additional type of function. This was not because I had any concrete examples of where internal functions were insufficient. It was more a fear that I would miss something in the design of the new language that would make it impossible for the users to adequately express their rules. This fear centered mostly on the recursion/iteration issue, since I had decided on a functional language with the knowledge that the users did not wish to use recursion. While the decision to use a functional language addressed a number of the important goals associated with the design of the new language, it also presented a dilemma. If I chose to use recursion, I would run the risk of providing the users with a language that they would find difficult and unpleasant to use. If I chose to use procedural control structures in the context of a functional language, there was a chance I would provide the users with a language that was not adequately expressive and perhaps difficult to use. As a result, I decided to provide a safety net to insure that the users had a sufficiently expressive language regardless of which option I chose.

To this end, I decided to continue to make the original language available to the users in the form of external functions. I did not, however, want to encourage the use of the original language. Nor did I want to alter the functional paradigm that I had selected. As a result, these external functions could be written in the original language and would have all the capabilities of the functions in the original language with one fundamental exception: assignments would not be allowed. They would be able to take arguments and compute values using whatever Tcl-based control structures were available. They would then be able to return a single value or a list of values that would have to conform to the syntax of the new language. These values would then be checked, at run-time, to insure that they were both syntactically and semantically valid given their placement within the rule or function. In this way, users could create functions that used procedural constructs and use them in the context of a functional language.

5.6.1.6. Expressions

Expressions were the fundamental building blocks of the new language. Expressions included primitive values, mathematical and logical equations, and control statements. Given the functional nature of the language, syntactically and semantically correct and complete expressions would always evaluate to a valid value in the language. In this way, complex mathematical and logical expressions could be built from simple expressions. These expressions, in turn, were used by rule statements to allow rules to set values on object.slots.

This section will discuss expressions in terms of the language's data types and control structures. It is not meant to be a language reference and will not have a fully defined syntax for the expressions. It will, however, delve into enough detail to illustrate the capabilities offered in the language. The discussion will primarily focus on the different language types supported and then touch on the control structures available. During the discussion of the language types, the operators that pertain to them will also be discussed.

5.6.1.6.1. Data Types

Although I had originally envisioned only two data types in the language, numeric and boolean, I soon realized that in order to provide the required functionality, I would need more types. In general, the types used in the rule language corresponded directly to types used in *RiverWare*. For example, the rule language included objects and object.slots. There were exceptions, however. Neither boolean nor string had analogous types in *RiverWare*. These were included to allow the rules to perform logical operations not available elsewhere in *RiverWare* and to print the state of rule executions. This section discusses each of these types, including why they were needed and how they were used.

5.6.1.6.1.1. Numeric

Numeric values were the primary types used in the rule language. This largely followed from the fact that *RiverWare* was a numeric simulation system that stored and worked with real numbers. The rule language followed *RiverWare's* lead and stored and manipulated real numbers. While no explicit

integer type was provided, the language allowed integers or real numbers to be used and converted the former into real numbers internally. As will be discussed in more detail in Section 5.6.2.3.2, each number had an associated scale and unit to fully qualify it and insure that it was used properly.

The rule language allowed for many of the standard numeric operators, including addition, subtraction, multiplication, division, integer division, modulo, power and unary minus. In addition, numeric values could be used with relational operators, which included greater than, greater than or equal to, less than, less than or equal to, equal to and not equal to.

5.6.1.6.1.2. Boolean

Boolean expressions were needed to allow for decisions within rules and functions. While it would have been possible to use numbers rather than boolean values in the antecedents of `IF` and `WHILE` control structures, this seemed to be a solution that benefited the language implementer at the expense of the language user for there was bound to be some confusion associated with the overloading of numbers in this way. The pre-defined boolean types were `TRUE` and `FALSE`.

In addition to the pre-defined types, boolean values also resulted from relational and logical operations. The relational operators were listed in the numeric value section and could also apply to DateTimes, which will be discussed in the following section. Logical operators included logical `AND`, logical `OR` and logical `NOT`. In addition, the operator `IN` was used to check for the inclusion of an item in a list using `item IN list`.

5.6.1.6.1.3. DateTime

DateTimes were needed to allow rules to make decisions and get object.slot values based on either absolute or relative dates and times. Absolute DateTimes could be specified from years to seconds. Relative DateTimes could be relative to other DateTimes or relative to the simulation clock. In

order to allow for sufficient flexibility to manipulate and use dates and times, the language allowed for many ways to express DateTimes. Table 1 contains a sample of some of these DateTimes.

Table 1: DateTime examples.

DateTime	Meaning Example of Use
`May, 1999`	May 31, 1999 24:00 `Mead.Outflow[May, 1999]`
`May 31, 1999`	May 31, 1999 24:00 `Mead.Outflow[May 31, 1999]`
`May 31, 1999 12:30`	May 31, 1999 12:30 `IF (CURRENT > May 31, 1999 12:30) THEN ...`
`CURRENT`	The current simulation timestep `Mead.Outflow[CURRENT]`
`PREVIOUS`	The previous simulation timestep `Mead.Outflow[PREVIOUS]`
`NEXT`	The next simulation timestep `Mead.Outflow[NEXT]`
`START`	The first simulation timestep `IF (CURRENT == START) THEN ...`
`FINISH`	The last simulation timestep `Mead.Outflow[FINISH]`
`TUESDAY`	Tuesday of the current simulation week and year. Days of the week would generally only be used for a simulation running on a daily, or possibly hourly, timestep. `IF (DAYOFWEEK == TUESDAY) THEN ...`
`JANUARY`	January of the current simulation year. `IF (MONTH == JANUARY) THEN ...`
`CURRENT MONTH, 1998`	The current simulation month in 1998. Thus, if the current simulation month were January, this expression would evaluate to January, 1998. `Mead.Outflow [CURRENT MONTH, 1998]`
`NEXT MONTH 15, 1998`	The 15th day of the next simulation month in 1998. Thus, if the current simulation month were January, this expression would evaluate to February 15, 1998. `Mead.Outflow [NEXT MONTH 15, 1998]`
`START MONTH + 2 MONTHS`	Two months after the simulation start month. Thus, if the simulation start month were January, this expression would evaluate to the last day in March for the current simulation year. `Mead.Outflow [START MONTH + 2 MONTHS]`
`CURRENT YEAR - 2 YEARS`	Two years prior to the current simulation year. `IF (CURRENT YEAR - 2 YEARS == START) THEN ...`

In addition to flexibility in expressing DateTimes, a number of operators were allowed on DateTimes. The last two examples show the plus and minus operators, which took DateTimes as operators and returned a DateTime. In addition, the relational operators, discussed for the Numeric type, could also be used with DateTimes.

5.6.1.6.1.4. Object

An object corresponded to an object in the *RiverWare* model. In general, objects needed to be combined with a slot name to make an object.slot, which was in turn used to get a value from the *RiverWare* model. The "`.`" operator was provided to combine an object with a slot name to make an object.slot.

5.6.1.6.1.5. Object.slot

An object.slot corresponded to an object.slot in the *RiverWare* model. It could be qualified, with either a DateTime or a row/column pair, to get a value from the *RiverWare* model. The syntax for this was square brackets following the object.slot. For example, `Mead.Outflow[Dec, 1998]` meant the value of Mead's outflow on December 31 1998 24:00 and `Mead.Outflow[]` was a special syntax that meant the value of Mead's outflow at the current timestep.
`Mead.EvaporationCoefficient[1,2]` meant the value of Mead's evaporation coefficient table at row 1 and column 2.

5.6.1.6.1.6. String

Strings were included to allow rule writers to use print statements for debugging. Strings were specified as text within quotes. The `CONCAT` operator was the only operator that took strings as arguments and was used to concatenate two strings into one. It could also be used with non-string types, such as Numeric or DateTime, which were converted into strings before being concatenated. This allowed rule writers to display the values of the rules' variables during execution.

5.6.1.6.1.7. Unknown

The unknown type was included to allow the language to contain incomplete expressions, which were in turn needed to allow the language's editors to save, load and display partially formed language elements. The ability to store and load partially formed rules and functions was included to give users the ability to create and edit their rulesets in a manner similar to the way they would create and edit rulesets using a traditional text editor. Requiring the users to store complete rules and functions seemed to be an undue burden and potentially dangerous, since it would likely lead to the use of placeholder values that might be overlooked when the rules and functions were reloaded.

5.6.1.6.1.8. List

Lists were heterogeneous, ordered groupings of any of the above types. I considered making the lists homogeneous, but was afraid that they would not provide sufficient flexibility. For instance, the users might need a list of object, numeric pairs to map between reservoirs and their storage volume. While this mapping could be represented using two homogeneous lists, this would be a less obvious solution and one that could lead to difficult to read code.

There were a number of operators that acted on lists. Table 2 contains a list of these operators and a brief description of their use.

Table 2: List operators.

Operator	Description
item IN *list*	Returns a boolean indicating whether or not *item* was found in *list*.
INSERT *item* INTO *list*	Adds *item* to the beginning of *list*.
APPEND *item* ONTO *list*	Appends *item* onto the end the end of *list*.
GET *type* AT *index* FROM *list*	Returns the item at the *index* position of *list*. *type* indicates the type of the resulting expression, which can be ambiguous otherwise.
REMOVE ITEM AT *index* FROM *list*	Removes the item from *list* that is at *index*.
SUBSTITUTE *item* AT *index* OF *list*	Substitutes *item* from the item at *index* in *list*.

Table 2: List operators.

Operator	Description
value1 TO *value2*	*value1* and *value2* must be the same type and be either Numeric or DateTime. If Numeric, this operation returns a list of values from *value1* to *value2*, inclusive. If DateTime, this operation returns a list of DateTimes using the current simulation timestep as the increment.

5.6.1.6.2. Control Structures

Control structures resembling those used in procedural languages were included in the language. However, instead of containing one or more statements that would act upon one or more variables, they evaluated an expression, perhaps multiple times, and returned a single value or list of values. In this way, they fit into the functional paradigm. The `IF` expression is a standard expression found in functional languages. The `FOR` and `WHILE` expressions, however, are not and were included to provide the users with a means of performing iteration in a familiar manner while still maintaining the language's fundamentally functional paradigm. They were to be used instead of recursion, which the users had already expressed strong reservations about using. This section discusses each of these control structures and gives simple examples of how they were used.

5.6.1.6.2.1. `IF` Expression

The `IF` expression was similar to the traditional, procedural *if* statement, in that it had an antecedent and one or two consequents. It differed from a traditional *if* statement in a number of respects. Rather than having any number of statements for consequents, it allowed only one expression per consequent. If there were two consequents, both were required to evaluate to the same type, since the `IF` expression returned the value associated with the evaluated consequent. Lastly, its use was determined by the context within which it was used. If it was used in a rule as part of a conditional assignment statement, either one or two consequents were permitted. If there was only one consequent and the antecedent evaluated to `FALSE`, no assignment would be performed. If, on the other hand, it was used within a function, two consequents were required in order to insure that the function would always return a valid value. An example of a simple `IF` expression that contains two consequents is in Figure

34. If the antecedent evaluates to `TRUE`, the first consequent is evaluated and its result (i.e., 5) is returned as the value of the entire `IF` expression. If the antecedent evaluates to `FALSE`, the second consequent is evaluated and its result (i.e., 7) is returned as the value of the entire `IF` expression.

```
IF ( <boolean expression> )
   2 + 3
ELSE
   3 + 4
ENDIF
```

Figure 34. `IF` expression example.

5.6.1.6.2.2. `FOR` Expression

The `FOR` expression differed from the traditional *for* loop in a number of respects. It did not iterate over a range of values; rather it iterated over a homogeneous list of elements. While these elements could be a list of ordered Numeric or DateTime values, they could also be any set of expressions. In this respect, it approximated the *foreach* statement found in Perl. Another difference was that it did not allow for the execution of procedural statements as it iterated through its list. Instead, it initialized a value at the start of the loop and at each iteration reset that value, if applicable, until each value in the list had been visited. It would then return a single value or list of values.

Figure 35 contains the outline for the `FOR` expression. Briefly, the loop variable is initialized at the start of the expression evaluation. Then, for each iteration of the `FOR` expression, the body is evaluated and the loop variable is possibly updated. After the last iteration of the `FOR` expression, the final value for the loop variable is returned.

```
FOR ( <type> <index name> IN <list> )
   WITH <type> <loop variable name> = <initialization expression> DO
   <body>
ENDDO
```

Figure 35. `FOR` expression.

Figure 36 shows a simple example of the `FOR` expression and will help in explaining how it is used. This example sums the odd numbers between 1 and 10, inclusive. Initially, the loop variable, `result`, is set to 0. Then the index variable, `index`, is set to the first value in the list and the body is

evaluated. Since `index` is equal to 1, the antecedent of the `IF` expression evaluates to `TRUE`, the consequent is evaluated and `result` is updated to `result + index`, which is 1. Next, `index` gets 2 and the body is evaluated again. Since the antecedent of the `IF` expression evaluates to `FALSE`, its body is not evaluated and the value of `result` is not updated. Next, `index` gets 3 and the body is evaluated again. Since the antecedent of the `IF` expression evaluates to `TRUE`, its body is evaluated and the value of `result` is updated to 4. This continues for each value in the list, with the odd numbers being added to `result` and the even numbers not being added to `result`. Finally, after the `FOR` expression has been evaluated for its last list element, the value in `result` (i.e., 25) is returned as the value for the entire `FOR` expression.

```
FOR ( NUMERIC index IN {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} )
   WITH NUMERIC result = 0
DO
   IF (index MOD 2 != 0) THEN
      result + index
   ENDIF
ENDDO
```

Figure 36. `FOR` expression example.

5.6.1.6.2.3. `WHILE` Expression

The `WHILE` expression was similar to the `FOR` expression in that it also contained a loop variable that was initialized at the beginning of the expression evaluation and possibly updated at each iteration of the loop. It differed in that the iteration of the loop would continue as long as the antecedent evaluated to `TRUE`, rather than iterating over a set list. Figure 37 contains an outline for the `WHILE` expression.

```
WHILE ( <boolean expression> )
   WITH <type> <loop variable name> = <initialization expression>
DO
   <body>
ENDDO
```

Figure 37. `WHILE` expression.

Figure 38 contains a simple example of the use of the `WHILE` expression. This example determines the amount of water that needs to be released from Lake Powell in order to insure that Lake Powell and Lake Mead have storage values that are approximately equal. It is a simplified version of the

functionality that was needed in one of the USBR's rules. Initially, the loop variable, `powellOutflow`, is set to 0. It is then used to determine both Lake Powell's and Lake Mead's storages. These storages are compared and while the difference between them is outside a certain tolerance, `delta`, the value of `powellOutflow` is incremented by `1.0`. If the storage values are ever within `delta` of each other, the iteration will cease and the current value of `powellOutflow` will be returned.

```
WHILE ( ABS (PowellStorage (powellOutflow) - MeadStorage (powellOutflow))
        > delta )
   WITH NUMERIC powellOutflow = 0
DO
   powellOutflow + 1.0
ENDDO
```

Figure 38. `WHILE` expression example.

5.6.2. Issues

Once the structure of the language had been decided, a number of additional issues had to be resolved. In general, these issues applied to the language as a whole. They included the use of comments, execution-time output, language safety and efficiency.

5.6.2.1. Comments

In order to make the language readable and maintainable, the language needed to support user-specified comments. Although one possibility was a highly flexible placement of comments, as is allowed in many procedural languages, a number of factors convinced me that this would not be appropriate. In part this depended on the programming environment that I had decided to use. As will be discussed later, I had decided to provide an environment that included individual editors for each of the major language elements with rules and functions being served by structure editors. In part this hinged on the structure of the language that encouraged relatively small and non-monolithic language elements.

The decision to use a structured programming environment meant that arbitrarily placed comments would lead to difficulty in specifying the language's syntax. Most languages have comments that are created and updated by free-form editors and are not included as part of the final, executable code. As a result, the syntax of these languages can be created in such a way that these comments are

ignored by the language compiler or interpreter. The new rule language, on the other hand, would need to read, display and allow for editing of these comments so that they could be manipulated by the structure editor. In the worst case, every possible location where a comment might appear would need a placeholder in the language syntax. Although it would be possible to implement a less extreme case, such as allowing for comments on their own line or at the end of each line, this did not seem to present a significant gain for the users and would have added significantly to the implementation time.

As an alternative, I decided to allow one comment per major language element. Thus, each ruleset, group, rule and function could have one, unrestricted comment associated with it. While this approach restricted the users in the placement of their comments, the language was structured such that these restrictions did not seem overly burdensome. There were, however, two cases in which I needed to determine if this solution was sufficient. One was related to rules, which could have any number of rule statements. As with rule statement constraints discussed previously, I explored the idea of allowing comments on a rule statement basis. Yet, given the anticipated increased visual complexity of the rule editor, I decided against this. Although this would restrict the users to a single comment per rule, this did not seem unreasonable, since presumably the rule was implementing a part of a single policy and this could reasonably be stated in a single, unrestricted comment. An alternative, which I did not pursue, was to include a comment statement. This would allow for comments to be placed between assignment and print statements without unduly adding to the complexity of the language.

The other case was with an `IF` expression within functions. Since a function could only have one comment, there could not be one comment for the first consequent and another comment for the second consequent. While this solution did not provide the degree of flexibility that the users might expect, I felt that an unrestricted comment could describe the nature of the function without resorting to comments about each individual line.

5.6.2.2. Output

In the initial language, the users had used print statements to debug their rules and functions. The users considered this to be a very useful feature. As a result, *RiverWare* was augmented to provide similar debugging capabilities for both rulebased and simple simulations. Given this, it was important that the new language provide the ability to print the status of ruleset executions.

As with comments, the ability to include print statements anywhere in the code would not be straightforward in the new language. The functional approach did not lend itself as easily as it had in the Tcl-based language to the arbitrary placement of print statements. Yet, given that the high-level executable language elements (i.e., rules and functions) were fairly self-contained, it seemed like there might be a reasonable alternative. As has already been discussed, `PRINT` statements were included as part of rules, which would provide the ability to print out the progress of the individual assignment statements. This seemed to provide the flexibility needed to allow the users to debug their rules.

Internal functions were another matter. Unlike rules in which there was a relatively straightforward and consistent means of placing print statements, internal functions did not have the notion of multiple statements. Instead, they consisted of a set of arguments, a single expression body and post-evaluation constraints. This allowed for the placement of print statements in two locations without making significant changes to the basic language structure. One was prior to the execution of the function and would display the name of the function being executed and the values of its arguments. The other was after the function had been evaluated and constrained, which would display the function's result. The user could select one of these, both of these or neither of these from the function editor. For completeness, this functionality was also added to external functions, even though they had the ability to make calls to print statements from within their body. While this would not provide the complete flexibility that arbitrarily placed print statements would, it would provide a great deal of information about the evaluation status of the functions being executed.

5.6.2.3. Safety

Safety was an important aspect of the language system that had not been adequately addressed in the initial language. There were two main areas where this was most noticeable and that caused significant and difficult to find errors. One was in the use of user-defined dependencies and the other was with respect to unit checking. Given that safety was an important design goal, these shortcomings needed to be addressed.

5.6.2.3.1. Dependencies

Dependencies were included in the original language to restrict the number of rule executions to only those that could affect the state of the simulation. If they were used properly, they would do so. Unfortunately, if they were not used properly, they would likely lead to incorrect results. And given the complexity of the rules, it was difficult to determine what the dependencies should actually be. To make matters worse, the system had no way of allowing the users to safely omit the use dependencies. They had to either specify them correctly or risk incorrect simulation results.

While I knew that I needed to fix this problem, I was unsure how I would do so. If I chose to omit the use dependencies, the speed of the simulations would suffer. Since performance was already an issue, this was not a good option. If I chose to use dependencies, they would have to be specified correctly to be of maximum benefit. Yet, requiring the users to specify them was not a good idea. In the original language they had both intentionally ignored them and unintentionally specified them incorrectly.

One last option was to have the users specify the dependencies and, rather than assuming they were correct, track all the object.slot accesses during a rule's execution. If a rule attempted to access an object.slot for which there was no stated dependency, an error would be generated. This seemed like a reasonable approach in that it would provide performance gains while still insuring correctness. However, there were a number of problems associated with it. In order to avoid as many run-time errors as possible, the users would have to conduct a careful analysis of the rules and functions, which could

be difficult and time-consuming. Given that the users had already shown a propensity to not use dependencies properly, this seemed unlikely to happen to the degree it should. The alternative, then, would be to run the simulation and allow the unspecified dependencies to be caught at run-time, which might take many executions to get the complete set of dependencies since it might be difficult to cover all paths the rules might take. And while this might eventually work, there would be no way of knowing for sure that all the dependencies were included, which would always leave open the possibility that a simulation might abnormally terminate. If an unspecified dependency were to be detected well into an hours long simulation, the users' frustration could lead to ill feelings about the system.

Although I did not like this option, it seemed better than the alternatives. Yet as I was considering how unspecified dependencies would be recognized, it occurred to me that instead of terminating the simulation, I could just add the unspecified dependencies to the rule's list of dependencies at run-time and continue the simulation. This was because the missing dependency could not have been accessed previously or it would have been recognized previously. This in turn meant that it had never been accessed by the rule and thus any previous changes to it could not have affected the execution of the rule. As a result, the fact that the dependency was not originally in the rule's list of dependencies had not altered the execution pattern of the rule and invalidated the simulation results.

This led to the conclusion that if I did not need to require the users to specify all the dependencies, I did not need them to specify any of the dependencies. Instead, all rules would be required to be executed at least once for each timestep. As they were executed, the language processor would keep track of the object.slot accesses and make these the rule's dependencies. If the values of any of these dependencies changed, the rule would be placed on the agenda for later re-execution. If any other object.slot values changed, the rule would not need to be re-executed because those object.slot values had not been used in the rule's previous execution and the fact that they had changed would not affect the rule's results. If the rule was re-executed and accessed a different set of object.slot values than had been accessed in the previous execution, only the new object.slot accesses would be used as the rule's dependencies. This was because the former set was for a path that the rule would not take again

unless one of its current set of dependencies changed. Thus, the list of dependencies associated with a rule would consist of all object.slots that had been accessed during the rule's previous execution. This mechanism had the dual benefit that it provided the performance gain associated with dependencies but did not require the users to specify them themselves.

5.6.2.3.2. Unit Checking

The use of values with different units was common for the rules written in the initial language. In general, values were combined and compared correctly, although since there was no verification of this, values with incompatible units were occasionally combined and compared. This resulted in rules with difficult to find errors and a language that was inherently unsafe. Therefore, a means of prohibiting the use of values with incompatible units was needed.

In order to do this, the language interpreter needed to keep track of the scale and units of each numeric value used by the rule. These values could come from user-defined literals, model-defined object.slots, function return values or as a result of values combined using arithmetic operators. In addition, the interpreter also needed to check each operation, whether arithmetic or relational, to insure that the operands were compatible for the given operator. If they were compatible, the operation would be allowed and the resulting value returned. If the result was numeric, a scale and units consistent with the original operands and operation would be associated with it. If they were not compatible, the operation would be prohibited and the simulation's execution would be terminated.

Keeping track of the scale and units associated with numbers was consistent with the approach taken by the rest of *RiverWare*. In fact, *RiverWare* allowed for an unlimited set of scales and units that could be associated with numbers. For example, a user could specify a flow, such as mead.outflow, with many scales and units, which included cubic-feet per second (*cfs*), cubic-meters per second (*cms*), *acre-feet/day* and *acre-feet/month*. For simplicity, however, *RiverWare* did not use the value, scale and units in the form specified by the user. Instead, it converted all values into metric units with a scale of 1.0. This internal representation was then used during all simulations.

The language needed to support the same set of scales and units that *RiverWare* supported. This would allow the users to write rules using the same values that they had used in their models. To this end, I decided to use *RiverWare's* approach to storing numbers and their values. Thus, regardless of the scale and units in which values were specified, they would be stored using metric units with a scale of 1.0. The only exceptions to this were dimensionless numbers, which were stored as specified. In addition to supporting a consistent underlying representation of numbers and allowing for faster execution of rules because all conversions were done up front, there was also the added benefit that the users could specify equations without regard to the scale and units of the values. Thus they could write an expression like `mead.outflow[] + mohave.outflow[]` and be assured that, if the two values had internally compatible scale and units, the addition would be performed accurately. For instance, if `mead.outflow` was specified in *1,000 acre-feet/month* and `mohave.outflow` was specified in *1.0 cfs*, the result would be valid since they are both flows and would be internally stored as *1.0 cms*.

Knowing the scale and units of each value was only the first step, however, in insuring that the rules performed only valid combinations and comparisons of values. The interpreter also needed to check each operation performed during a rule's execution to insure that its operands were compatible. In some cases the operands had to have the same underlying units to be combined or compared. These included addition, subtraction and the relational operators. In other cases, including multiplication and division, the underlying units could be different from each other. While the rules for combining and comparing values were well established, the mechanism for efficiently doing these needed to be determined.

The mechanism chosen stored and tracked the components associated with a number's unit. These components, which included length, mass and time, would be stored as a 'signature' that represented the unit. The signature would be a series of numbers, one for each component, which represented how many of each of the components (i.e., their power) were included in the unit. For instance, cubic-meters per second (*cms* or m^3/s) would contain three lengths (cubic meters), zero masses

and minus one time (seconds) and would have a signature of 3:0:-1. The negative value was used to denote the fact that the seconds were in the denominator.

These signatures could be used during relational and arithmetic operations to check whether the values were compatible for the given operation. Thus, the equation 4 *cms* + 5 *cfs* would be allowed since both of its operands' signatures were 3:0:-1. On the other hand, 4 *cms* + 5 *m* would result in an error, since its operands' signatures were 3:0:-1 and 1:0:0, respectively, and addition requires that the underlying units be the same. Relational operators worked similarly to addition and subtraction in that the signatures needed to be the same for the comparison to be performed. Multiplication and division, on the other hand, had to be treated differently, since they allowed for the combination of values with different units. For example, m^2 * 2 *m*, which would evaluate to 2 m^3, and 4m^3 / 2*s*, which would evaluate to 2 *cms*, would both be valid operations.

For multiplication, the units of the result were computed by multiplying the units of the operands. For example, m^2 * *m*, which evaluated to m^3, has signatures 2:0:0, 1:0:0 and 3:0:0, respectively. Notice that this amounted to the addition of the individual components of the values' signatures, which worked given that the signatures held the power of each component and multiplication resulted in the powers of the units being added. Division, on the other hand, computed the units of its result by dividing the operands units. Thus, m^3 / *s*, which evaluated to *cms*, has signatures 3:0:0, 0:0:1 and 3:0:-1, respectively. Unlike multiplication, in which the components of the signatures were added, the signatures for division were subtracted. Again, this worked because division caused the units of the second operand to be placed in the denominator, which in the signature notation meant it was given a negative value.

This mechanism of storing and tracking the scale and units of each value used in the rules allowed the rule writers to combine and compare values without having to be concerned with whether or not they were compatible. This made for a language that was considerably safer than the original language and allowed the users to concentrate more on the details of the policy being represented. The

use of this mechanism also helped to make the language more readable. The original language had required the users to use the suite of `GetValue` routines, which required the specification of the requested value's scale and units to access object.slot values. In addition, the users had to make use of various conversion routines whenever one value was not directly compatible with another. Both of these added details to the rules that were not directly related to the rules' underlying policies.

5.6.2.4. Efficiency

One of the main reasons for abandoning Tcl was that it was slow. Its speed was such that the execution of rules took a large proportion of time for rulebased simulation runs. In one performance test of a typical model and ruleset, the execution of the rules accounted for 57% of the total simulation time. Much of this could be attributed to the fact that the rulebased simulation controller solved the system in an iterative manner by switching between the mass balance simulator and the execution of the rules, which resulted in each rule being called many times for a given timestep. While this inefficiency needed to be addressed, this algorithm was part of the rulebased simulation code and I had no control over it. As a result, in order to improve the performance of the rulebased simulation runs, I needed to see if I could improve the performance of the language interpreter.

This desire to increase the speed of the simulations helped motivate the decision to create a custom language. To this end, I explored the programming language literature and I found that it was possible to add performance enhancing features to the language that tracked and executed only that code that needed to be [Hudson, 1991]. For instance, a function without arguments could be executed once per timestep and that result used whenever the function needed to be re-executed. In this way the time taken to execute the function would be incurred just once and, if the function was called many times, the speed of the simulation could increase significantly. This mechanism was similar to the way rules were only executed if changes to the state of the *RiverWare* model made it possible for them to arrive at different results from their previous execution. The main difference was the finer degree of tracking that was possible. In fact, it would be possible to track changes and limit executions on an expression level,

if desired, although the overhead involved with this fine a level of tracking might offset any performance gains.

While this mechanism could provide significant performance gains to the execution of the rulebased simulations, there was no time to implement any of these features. As a result, the early implementations of the language were slower than the initial, Tcl-based language. When one of these mechanisms, single execution per timestep of argument-less functions, was later added to the language, the language equaled the speed of the Tcl language. Although I suspect that extending this mechanism to certain of the non-conditional, terminal expressions might also yield performance improvements, I am no longer involved with the project and am unable to make these changes.

5.6.2.5. Execution Order

The order of execution of the rules within a ruleset was an issue that I was under the impression was adequately addressed. In the former system, the rules were individually prioritized without respect to the group in which they resided. As it turned out, discussions with the users made it clear that another method of prioritization was needed. This need became clear during a walkthrough of the new language design and was, in part, motivated by the new language's ability to group rules. In addition to the ability to prioritize rules individually, a subset of the users also wished to prioritize the groups themselves and rules within the groups. Thus, one group would have a higher priority than another and, as a result, all of its rules would be executed before the rules of the other group. Given that there were two camps of users, each of whom felt strongly that they needed to be able to specify the order of execution in one of these two manners, I added an optional priority field to the policy group specification. In addition, I added a priority type field to the ruleset in order to record the desired type of rule prioritization.

5.7. Programming Environment

Once I felt I had a good understanding of the overall structure and direction of the language, I set about to determine how I could integrate the programming environment into the system. I knew that

I needed an environment that provided more assistance to the users than the original language. I also needed an environment that would address many of the problems that existed in the old language. An unstructured, text-based editor, while providing the basic functionality needed to create and maintain rulesets, would be unlikely to provide the support necessary to attain these goals. This had been amply demonstrated in the initial language system that had little environmental support and in which many users found the creation, maintenance and readability of the rules difficult. And even though many of the difficulties could be addressed in the context of an unstructured, text-based programming environment, a more structured environment would be more appropriate.

The design of this structured environment was based on the underlying language elements. Two of these elements, rulesets and groups, were essentially containers that were to be filled with other language elements. The other elements, rules and functions, were more fundamental language elements that contained the logic of the policies being represented. The former two language elements both made use of what I will call container editors, while the latter two language elements made use of structure, or syntax-directed, editors. The following sections will discuss the editors and the rationale for their designs.

5.7.1. Container Editors

The container editors were intended for the creation and editing of rulesets and groups. Their functionality was similar because the characteristics of these two language elements were similar. One of the main thrusts in the design of these editors was to shield the users from the underlying operating system and many of the design decisions were predicated on this need. In addition, these editors also needed to provide the basic functionality needed to view and update their underlying language elements. These latter needs, in many cases, overlapped with the former. The following two sections will discuss the ruleset and group editors individually.

5.7.1.1. Ruleset Editor

The ruleset editor was used to view and edit ruleset contents and properties. See Figure 39 for an example of a populated ruleset editor. This section will discuss the ruleset editor and the functionality it provided.

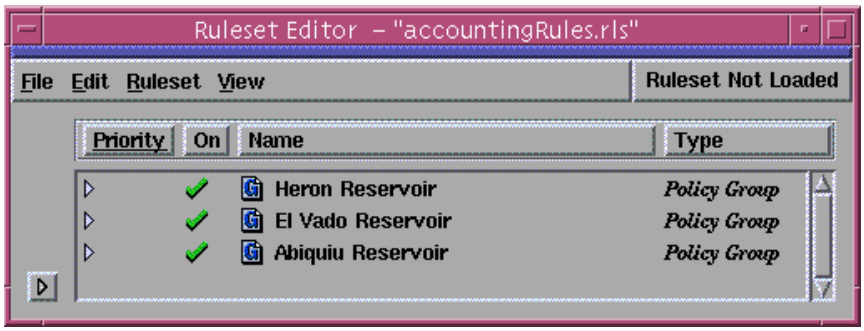

Figure 39. Ruleset editor.

5.7.1.1.1. Ruleset

The ruleset editor included the functionality to allow users to create, open and save rulesets. In the former system, the use of the operating system was required, which was difficult for some of the users. Ruleset creation was one of the main areas that troubled the users because it required them to either create a ruleset from scratch or copy an existing ruleset and modify it. Both of these operations had to be performed outside of the context of the programming environment and required at least a modest amount of skill using a UNIX shell. In addition, it required the users to leave the context of *RiverWare* to create something that would be used exclusively by *RiverWare*.

To address this problem, the new system allowed the user to create an empty ruleset that was, until saved, unassociated with any particular UNIX file. Users could create as many rulesets as desired, with each ruleset having its own editor. In addition to the creation of rulesets, users could also open existing rulesets for editing. This functionality was the same as that offered in the former system and provided users with a file chooser that they could use to navigate the UNIX file system. While it would have been possible to provide a means for shielding the users completely from the file system, the use of a file chooser was common to most commercial software programs and thus not considered an undue

burden. Similarly, saving of rulesets required the use of the file chooser. This was a departure and improvement over the former system in which the users were unable to save rulesets from within the programming environment.

5.7.1.1.2. Editing

In addition to creating and saving rulesets, users also needed to be able to manipulate the contents of their rulesets. This included the ability to add, remove and alter the groups in a ruleset and the ability to copy or move a group from one ruleset to another. In the former system, these actions had been accomplished by directly manipulating the ruleset, rule or function files or by copying or moving files among the directories that served as groups. Given that the new system was based on a single ruleset file in which all parts of the ruleset resided, it would be unreasonable to require the user to have to manipulate the file directly in order to extract or insert a group, rule or function. In fact, it would defeat the goal of insulating the user from the operating system. Therefore, there needed to be a means of creating and manipulating individual components of the ruleset.

To this end, the editor provided options to create empty and generically named policy and utility groups in a ruleset. A policy group could in turn have one or more rules or functions added to it, while a utility group could have one or more functions added to it. These rules and functions needed to be defined and referenced to be used during a rulebased simulation. In addition, groups, rules and functions would most likely need to be renamed from their generic default to something more meaningful. The removal of groups, rules and functions was similarly easy; the user had only to select the element to be removed and invoke the remove option.

In addition to adding elements to and removing elements from rulesets and their groups, the ability to rename, copy and move these elements also needed to be provided. The renaming of a language element only involved a single ruleset and the editor provided for the in-line editing of the names of the groups, rules and functions. To do this, the user would double click on the name of the element to be renamed and type the new name in the provided text field. Copying and moving language

elements, on the other hand, could involve either a single ruleset or multiple rulesets. For instance, a user might want to move a function from a policy group to a utility group in the same ruleset or might want to copy a group, with all of its contents, from one ruleset to another. To cut and copy a language element, the user would select an item with the mouse and use a menu item or function key to either move or copy the selected item to an internal buffer. To paste this item, the user would select an appropriate item into which the element would be placed and use a menu item or function key to copy it from the buffer to the selected item. In addition to the menu and keyboard driven options, the ruleset editor also allowed the user to click and drag an element from one location to another. As with creation above, the selected item would have to be of a compatible type. For instance, a rule could not be pasted into a utility group, although it could be pasted into a policy group. In addition, an item could not be pasted into an element if it would result in a naming conflict.

5.7.1.1.3. Prioritization

The original programming environment had allowed users to reprioritize their rules. This was done by selecting the priority number and changing it. While this worked reasonably well, it was a little cumbersome. To make this operation easier, the new environment overloaded the move operation described above. In addition to placing an item in a particular group or ruleset, it also specified the priority of the moved item relative to its neighbors. For instance, the rules in a group could be individually prioritized in order to set their order of execution within the group. To re-order the priority of the rules, the user had only to drag and drop a rule into a new position relative to the other rules in the group. The resulting visual order would represent the relative priorities of the rules.

5.7.1.1.4. Viewing and Navigation

The high-level elements in a ruleset were groups and were represented in the ruleset editor by a single line per group. In order to provide access to the elements in the groups, a treeview structure, like that used by the *Macintosh* window manager, was used. This would allow the users to toggle between viewing just the group's line and viewing the group's line and its contents. The contents, which could include rules and functions, were listed on their own lines underneath the group to which they belonged.

For an example of this, see Figure 40. In addition to opening a group in order to view its contents, users could also open an editor associated with a selected language element. This could be accomplished by double clicking on the line associated with the language element or by selecting the element's line and using a menu option to open the element's editor.

In addition to a ruleset's groups, the ruleset editor needed to support the editing and viewing of its description text, method of prioritizing rules and precision with which numbers were displayed by rule and function editors. For an example of this, see Figure 41. Note that this information is hidden by default, but can be accessed for viewing and editing by pressing the button with the triangle located by the lower left hand corner of the group list.

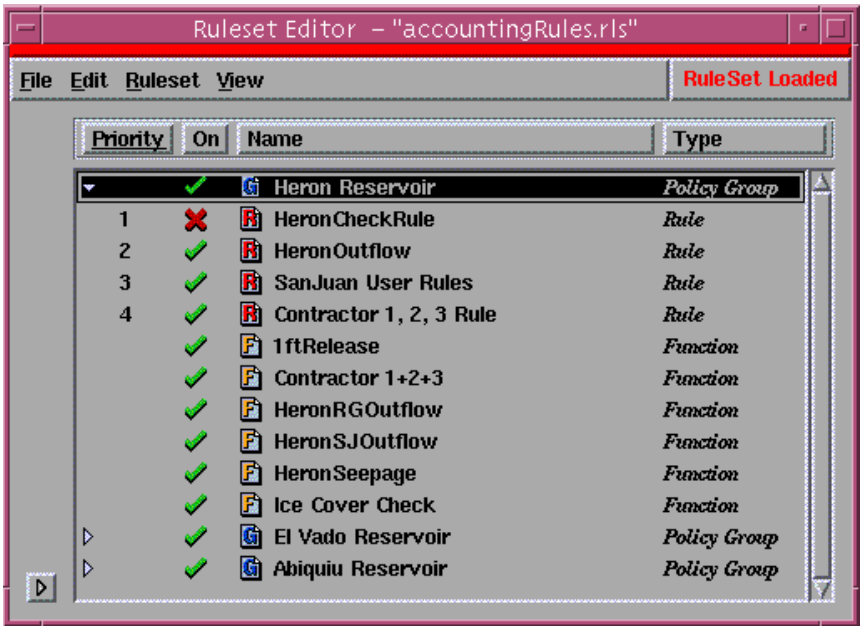

Figure 40. Ruleset editor with an open group.

5.7.1.2. Group Editors

Group editors were similar to ruleset editors in that they were essentially containers. Instead of holding groups, however, they held rules and functions. Yet, given that the ruleset editors allowed for viewing and editing of groups and their contents, the functionality of the two editors overlapped considerably. In fact, aside from the fact that group editors were specific to groups and had their own

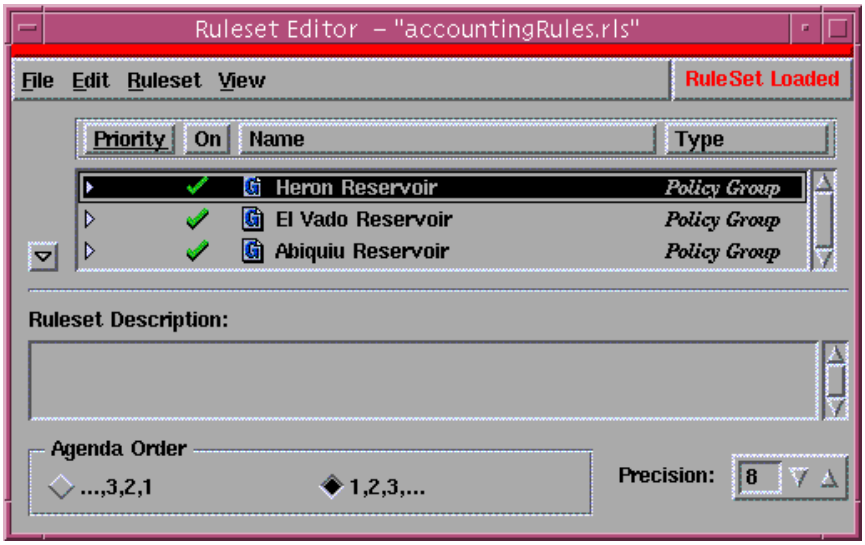

Figure 41. Ruleset editor with "hidden" information visible.

description text, there was no functionality that was unique to the group editors. Figure 42 contains an example of a group editor. Notice that the contents of the group editor are the same as in the corresponding opened group from Figure 40.

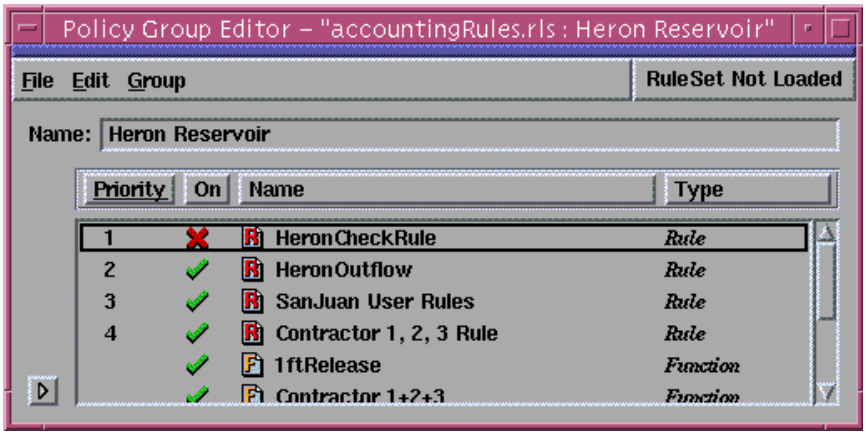

Figure 42. Group editor.

5.7.2. Structure Editors

The structure, or syntax-directed, editors were motivated by the need to provide a high level of support for both reading and writing of rules and functions. Structure editors could address these issues by controlling the formatting of the code as it was written and restricting the rule writers to only the

syntactically valid language options during the creation and updating of their rules and functions [Neal and Szwillus, 1992, Minör, 1992, Welsh and Toleman, 1992, van de Vanter, Graham and Ballance, 1992, Bahlke and Snelting, 1992, Göttler, 1992, Bellamy and Carroll, 1992, Lerner, 1992, Halewood and Woodward, 1988, Tennant, Ross and Thompson, 1983]. While neither of these would insure that the users produced code that accurately reflected their policies, it would make their code easier to read, easier to understand and syntactically correct. This section will discuss some of the motivations for using structure editors and then discuss how they were used in the programming environment's rule and function editors.

5.7.2.1. Readability

While I had hoped that the structure of the new language would help to improve the readability of the language, I knew from my experience with the initial language that this alone would not be sufficient. The users had sometimes been careless with the creation and editing of their rules and functions and had often taken what could have been fairly readable rules and made their meaning difficult to discern. Often this was caused by rules or functions that were poorly structured and sometimes monolithic. Other times it was caused by an inappropriate use of language features, such as global variables. Lastly, the users' use of formatting was often unstructured and, at times, seemingly random. This latter point, while perhaps not catastrophic, did impede the readability of the rules to a large degree. And while the rules would execute as intended, this was only one objective of the rules. For while the computer needed to be able to read and interpret the rules, human users also needed to do so. The first two of these problems would be largely or completely mitigated by the new language. The last one, however, would not be without some support from the language or the programming environment.

The language could have enforced some degree of formatting on the rules and functions. For example, the language could use indentation as a part of its syntax, since indentation can help with program comprehension and debugging [van Laar, 1989, Gilmore and Green, 1988]. This is the approach taken by *Python*, an object-oriented scripting language [Lutz, 1996]. This option held some

appeal in that it would help increase the readability of the resulting code. Yet, this solution was aimed at a completely text-based language that relied on the use of traditional text editors. These editors would not, in general, address the writability issues that were previously discussed. In addition, the use of a purely language-based approach would, outside of a markup language, preclude the use of other visual cues, such as font type, style and color. As will be discussed in Section 5.7.3.1, I wanted to provide the users with the ability to edit their rulesets outside the context of the structure editors and the use of a markup language would make this difficult. Given these considerations, I decided that a purely language-based solution to formatting, while by no means harmful, was not appropriate for the new language. As a result, I decided to use the programming environment to improve the readability of the users' code.

The programming environment could, through the use of a structure editor, format the language elements that were part of the rules and functions. This formatting would create a consistent and organized appearance, improving the readability of the code [Kellogg, 1987]. For example, an `IF` expression could be indented and spaced in a consistent and readable manner. Figure 43 contains an example of a formatted `IF` expression within a function editor. Similarly, numeric expressions could be formatted to display the equations symbolically, rather than linearly. Figure 44 contains a simple example of this. This consistency of formatting would take away some of the ambiguity that might be associated with a particular user's style. While some might argue that this was overly restrictive, I felt that providing a common means of formatting rules would insure that there would be no confusion associated with inconsistent indenting and spacing. This would free the users to put their creative energy toward the creation of rules that reflected their policies.

As mentioned previously, a structure editor could also include visual cues in addition to indenting and spacing. For instance, certain language features could be displayed using either special fonts or colors [Oman and Cook, 1990, van Laar, 1989, Baecker and Marcus, 1986]. This use of visual cues could be used to enhance both the readability and writability of the rulesets. For instance, an object.slot reference, such as **mead.outflow[]**, could be displayed in bold Arial, while a similarly named

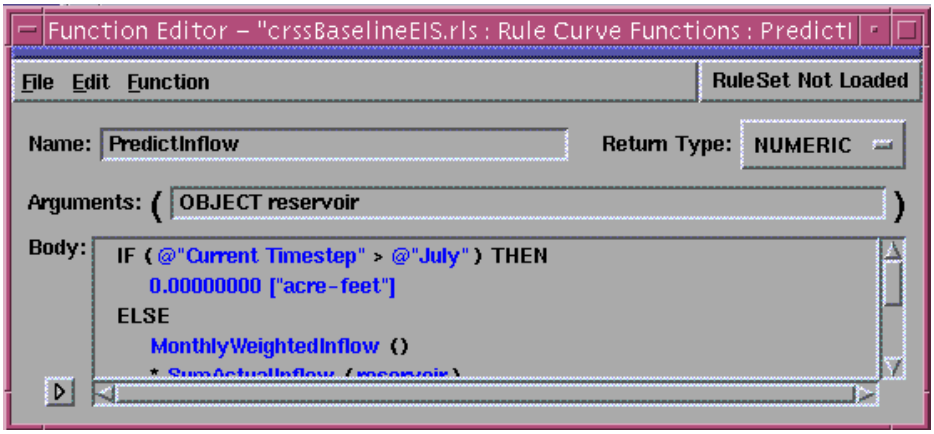

Figure 43. Formatting of an IF-THEN-ELSE expression.

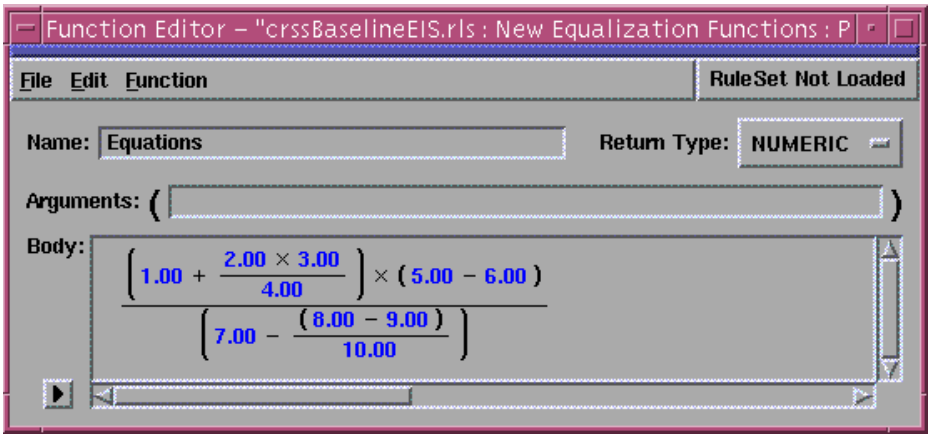

Figure 44. Formatting of an arbitrary numeric expression.

function, `meadOutflow()` could be displayed using plain Courier. This would help to avoid the confusion that might result from similar names.

During editing, placeholders were used to show the users where they needed to add code. For example, Figure 45 contains a partially completed conditional expression, in which the placeholders for expressions are enclosed in angle brackets. By using different fonts, styles and, possibly, colors, the rule writer could more easily discern that the antecedent takes a different type of expression than the first consequent. Note that, in this example, the fact that the second consequent is a numeric expression means that the first must also be.

```
IF ( <boolean expression> ) THEN
   <numeric expression>
ELSE
   mead.outflow[]
ENDIF
```

Figure 45. Partially constructed conditional expression.

5.7.2.2. Writability

As with readability, the writability of the language could be improved by the use of a structure editor. A structure editor could be used to show the user what options were available as a rule or function was being created. While it would not guarantee that the rule or function was semantically correct, it could be used to insure that it was syntactically correct. In addition, it could be used to help novice users learn the language and know which system and user-defined functions were available for use in any given expression. The following sections discuss how writability was addressed by the rule language's structure editors.

5.7.2.2.1. Selection and Palette

The structure editors worked by associating a selected expression with a set of valid substitutions. This selection could be a simple expression such as a number or a placeholder. It could also be a complex expression such as a boolean relational expression. In either case, the user would only be provided with those substitutions that would insure the rule or function remained syntactically valid. The set of substitutions were placed on a palette that contained all possible substitutions for any given selection with only the valid ones enabled for the selected expression. Thus, the user would know both what was allowed and what was not allowed for any given selection. Figure 46 contains an example of a function editor with a numeric expression selected and Figure 47 contains an example of the palette with the valid substitutions enabled for this selection.

For a more complete explanation, Figure 48 contains an incomplete conditional expression that will help illustrate how a selected expression and the palette worked together. If `<boolean expression>` were selected, the user would be restricted to using a boolean expression for the antecedent. These would include boolean literals and functions and relational operators that returned

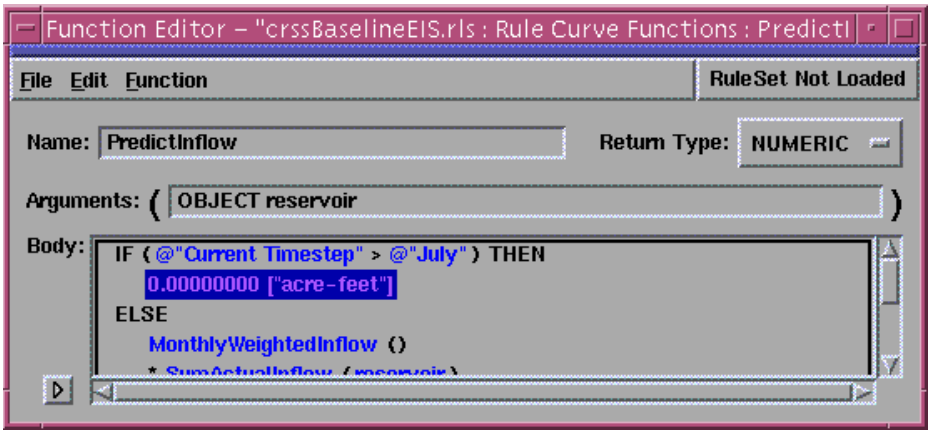

Figure 46. Function editor with an expression selected.

boolean values. If one of the consequents (i.e., one of the `<expression>`s) were selected, the user would not be restricted to any particular type, at least not initially, because a conditional could evaluate to any type of expression. The only restriction would be that both consequents would have to be the same type. Thus, if one consequent had been specified, the other would be constrained to the same type.

One point about the structure editing environment that is evident from Figure 48 is that an expression can be syntactically correct without being complete. This was needed to allow users to create rules and functions in a top-down fashion. While it could have been possible to create a structure editor in which only bottom-up editing was allowed, such an editor would not be very useful. For example, consider an `IF` expression in which it would be difficult to provide an intuitive framework in which the statement could be created with the antecedent and consequents already filled in. While it could be done, its use would likely negate many of the benefits associated with the structure editor. Another benefit of allowing partially complete expressions was that rules and functions could be saved and reloaded without being completely specified.

5.7.2.2.2. In-line Editing

While the palette was useful for helping users create and edit their rules and functions, there were times when its use was not warranted. One of these times was the addition of literals to an expression since it would be impossible for the palette to contain all possible literals. Also, in some cases, the users would know exactly what expression they wished to add and could do it faster directly

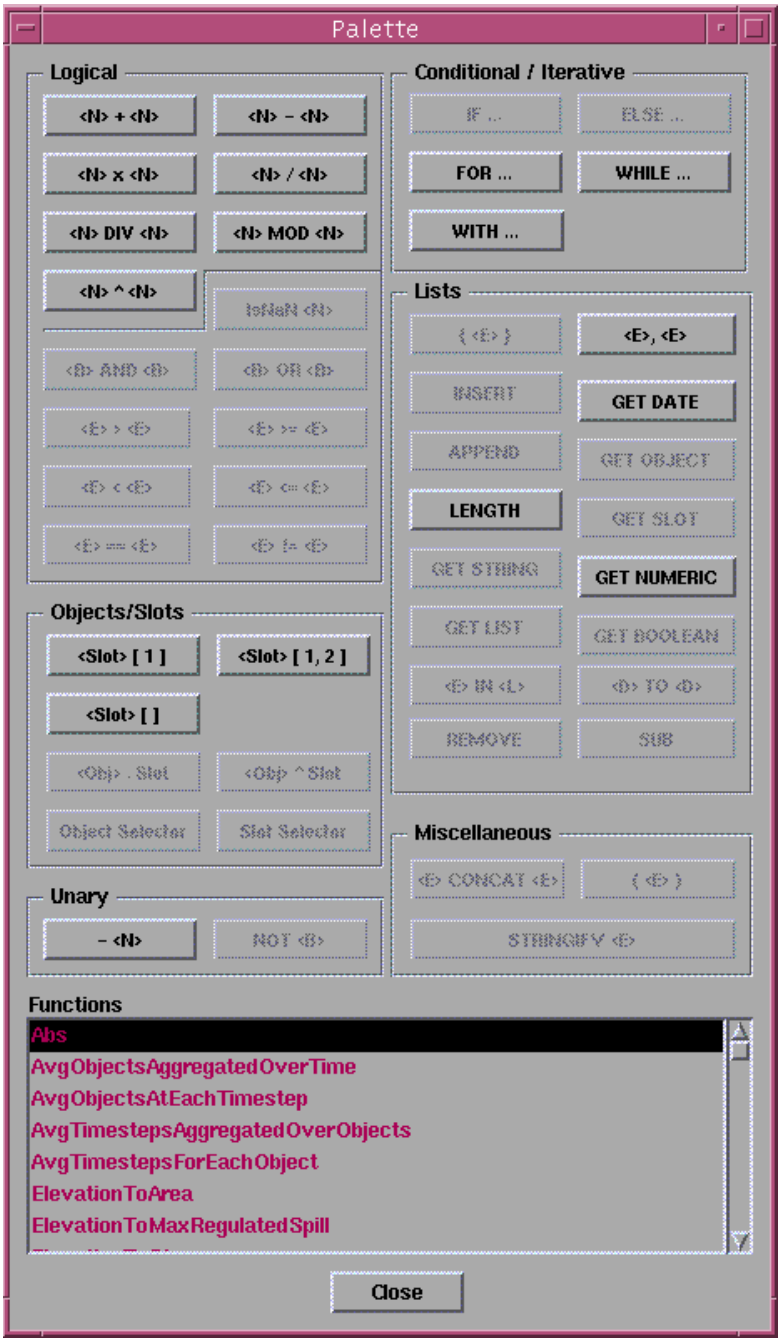

Figure 47. Palette with valid substitutions enabled for selected numeric expression.

as opposed to using the palette. While a text field could have been added to the palette to support this functionality, this seemed unnecessarily indirect. Instead, the structure editor allowed for in-line editing of expressions. It worked in the following manner. When users wished to edit an expression using the in-line editor, they would double click on the expression or select the expression and chose the appropriate menu option. This would cause a text field to be opened and populated with the text from

the selected expression. This expression could then be edited by re-typing or changing the selected expression. When finished, users would hit the carriage return and the new expression would be parsed to insure that it was syntactically correct. If it was, the newly created expression would be integrated into the existing expression with whatever formatting was necessary. If it was not, the user would be alerted to this fact and asked to re-enter the expression. For an example of a function editor with an in-line editor opened, see Figure 49.

```
IF (<boolean expression>) THEN
   <expression>
ELSE
   <expression>
ENDIF
```

Figure 48. Incomplete conditional expression.

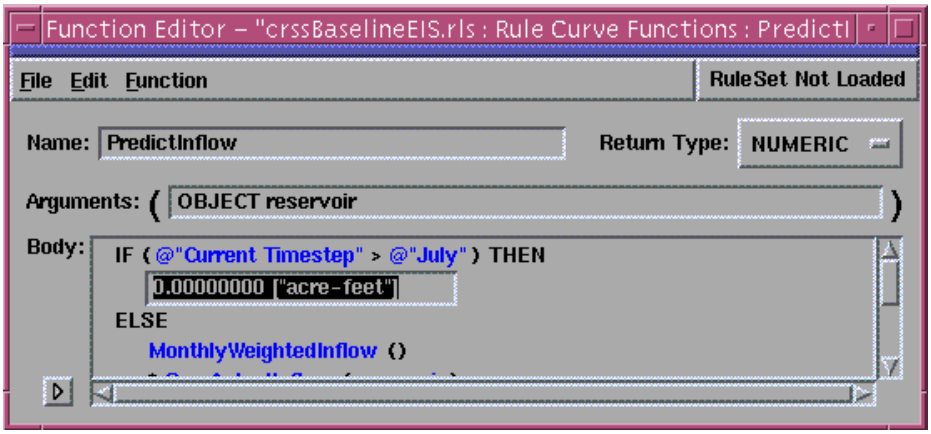

Figure 49. Function editor with in-line editor open.

5.7.2.2.3. Cut, Copy and Paste

In order to provide the editing support that the users would need to create and modify their rules and functions, the structure editors needed to support basic editing functionality. This functionality needed to be provided in the context of the structure editor and needed to allow the users to share expressions between the elements of whatever rulesets they currently had open. The most obvious approach was to provide cut, copy and paste operations that shared a buffer that was available to all rulesets. The cut and copy operations could be unrestricted in that they could apply to any selected expression and would place the selected item in the common buffer. The paste operation, on the other

hand, would need to be restricted since the placement of expressions was restricted to insure that the overall expression remained syntactically valid. The mechanism for supporting cut, copy and paste operations was the use of the standard menu and keyboard operations provided by the majority of the commonly used editors and word processors.

The buffer for the cut and copied expressions was invisible to the users. In general, this worked well since the users usually cut/copied an expression and immediately pasted it. Originally, however, I had intended on providing a visible cut/copy buffer that would contain a predefined or user configurable number of the last cut or copied expressions. I felt that this could be used as a recent history of changes and, if desired, a workspace to construct expressions before placing them in their rules and functions. I felt that this would be a useful facility given that I was not sure how easy it would be to use the structure editor. Time constraints, however, prohibited the inclusion of this functionality and, as it turned out, the users did not seem to need it.

5.7.2.3. Rule Editor

The rule editor was used to create and edit rules and their associated rule statements. The rule editor was similar to the ruleset and group editors, in that it contained a list of items that could be added and removed using menu options and function keys. It also supported the cutting, copying and pasting of the rule statements and their associated expressions. And like the ruleset and group editors, newly created rule statements were initially incomplete. They had to have their placeholders specified by the user to be of use in a simulation. See Figure 50 for an example of a rule editor with two rule complete rule statements and see Figure 51 for examples of the incomplete rule statements.

The main difference between the rule editor and the ruleset and group editors was the manner in which the rule statements were edited. For while the groups, rules, and functions that were part of the ruleset and group editors each had their own editors, rule statements needed to be edited within the context of the rule editor. To do this, the structure editor was needed. The reason for this was that while the rule statements themselves were not expressions, they made use of expressions. And since

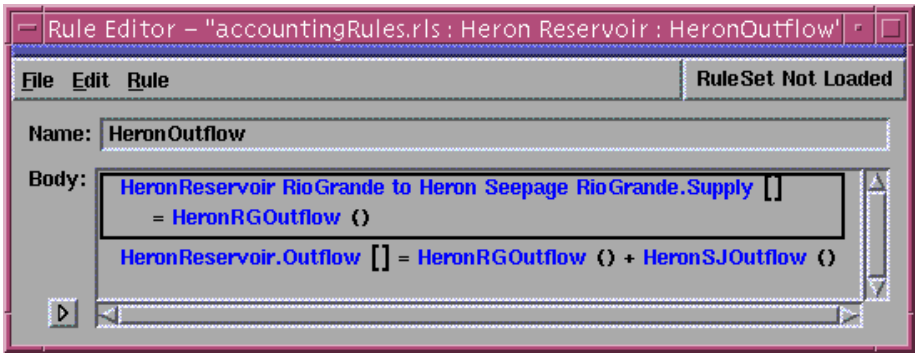

Figure 50. Rule editor with two complete rule statements.

expressions needed to be edited using the structure editor, the rule statements also needed to be edited using the structure editor. While it may seem arbitrary to state that rule statements were not expressions and only made use of them, there was a good reason for this distinction. Rule statements, like rulesets, groups and rules, were considered specific to the rule language and not needed outside of it. Expressions, on the other hand, were designed to be used both by the rule language and, eventually, in other parts of *RiverWare*. Examples of the latter case were Expression slots, which were used to compute values during rulebased and non-rulebased simulations.

```
<object.slot expression>:= <expression>
<object.slot expression>:= IF (<boolean expression>) THEN
                               <numeric expression>
                           ENDIF
PRINT <expression>
```

Figure 51. Incomplete rule statements.

In addition to the rule statements, the rule editor also needed to support the editing of the rule's description and its execution constraint. The former was addressed in the same way it was for the ruleset and group editors. The latter, as may be recalled, was an expression that needed to evaluate to `true` in order for the rule to be executed. In order to allow for this, the rule editor contained a simple expression, the constant boolean expression `TRUE`, that could be edited to contain any boolean expression. If the user did not edit the expression, the rule would always execute. If the user changed the expression, the rule would only execute if it evaluated to `TRUE`. Like all other expressions, this one was editable using the structure editor and thus guaranteed to be syntactically correct.

In addition to the editing capabilities provided by the rule editor, the user could also invoke the function editor for a particular function by selecting and double clicking on the name of the function in an expression. This enabled the user to drill down and explore the functions used by a rule and by other functions.

5.7.2.4. Function Editors

As was discussed in Section 5.6.1.5, two types of functions were available in the language, internal and external functions. The main distinction between the two was the underlying language of their bodies. Internal functions were specified completely in the new language using a single expression for their body. External functions were specified in the new language with the exception of its body, which was written in the old language. Given both the similarities and differences, it made sense to provide two editors that shared a great deal of functionality. In fact, the two editors differed only in how they supported the editing of the bodies of their functions. The following sections will discuss the different parts of these functions and how editing was supported in each.

5.7.2.4.1. Body

Internal functions used a single expression from the new language as their body. This expression could be anything from a placeholder to a complex, conditional expression. Since an expression was used, the structure editor was used as the basis for all editing. In this respect, its editing capabilities were the same as those discussed for the rule editor. See Figure 43 for an example of an internal function editor.

External functions, on the other hand, used the old, Tcl-based language as their body. While a structure editor could have been used to support editing the free-form text in the body of external functions, it made little sense for a number of reasons. One was that it would have taken a great deal of time to develop the necessary functionality. Another was that the intent of these external functions was to allow users to create small functions to handle functionality that either could not be expressed in the new language or was sufficiently difficult that it made more sense to express it using the old language.

Accordingly, I did not wish to encourage their use. Lastly, the point of the structure editor was to aid novice users and the point of the external functions was to aid experienced users, so a structure editor would not be necessary or even wanted. See Figure 52 for an example of an external function editor.

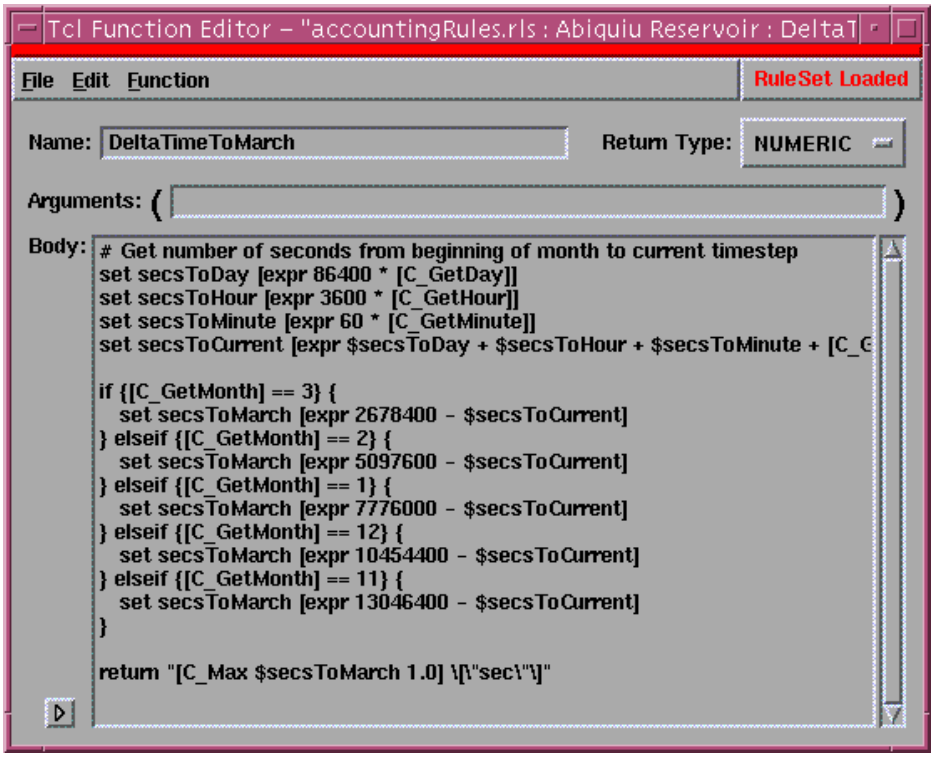

Figure 52. External function editor.

5.7.2.4.2. Return Type

The return type of a function was needed to allow the structure editor to know which functions could be used as valid substitutions during the editing process. Initially, I had assumed that the internal function editor would deduce and set the return type of the function without input from the user and that the external editor would require the user to specify the return type. As a result, I designed the function editors to include an option menu that contained all possible return types, including one for the unknown return type. A newly created function would default the return type to the unknown type. For external functions, the user would be responsible for setting the return type manually.

For internal functions, the editor would determine the return type as the user built up the body. It would also update the return type if the body of the function was ever reset. While this worked, it soon

became clear that having the editor determine the return type was too limiting. The main reason for this was that the users would sometimes create their functions without wanting to immediately populate them. This would leave the function with an unknown return type, which would make it unavailable for use in other expressions, since no expressions can use functions that returned the unknown type. To get around this restriction, the users would have to place a simple expression, such as a "1", in the body of their functions in order to get the expression to be of the desired type. While this worked, it was clumsy and also had the potential of allowing users to inadvertently run a simulation with an incorrect function definition.

To resolve this problem, I combined the functionality provided by the external editor with that of the internal editor. This allowed the user to explicitly set the return type or allowed them to have the editor determine the return type as the function was created and updated. The remaining restriction was that the user could not change the return type to something that was incompatible with the body of the function. In this way, the user could create incomplete, typed functions while still allowing the system to determine the return type by default.

5.7.2.4.3. Parameter List

The function's parameter list consisted of a comma-separated list of type name, variable name pairs. See Figure 43, which has a single argument in its argument list. Initially, I had intended on using the structure editor or a more structured approach in the function editor itself to allow users to populate the argument list. Unfortunately, time did not permit this and, as an alternate solution, I had users type the parameter list directly. Then, upon hitting the carriage return, the user-specified parameter list was parsed to insure syntactic correctness. If the parameter list was not syntactically correct, users would be alerted and required to update the string. While this worked reasonably well, its use was at odds with the underlying motivation for using a structure editor, since it required users to know and manually specify part of the language.

5.7.2.4.4. Post Evaluation and Diagnostics

Post evaluation checks were used to inspect the value to which the function evaluated and either reset that value before returning it or terminate the simulation. Any number of these checks were permitted as part of the function and, as with rule statements, they used expressions but were not themselves expressions. As a result, the adding and removing of these checks was accomplished outside of the structure editor using menu options and function keys. Their form is shown in Figure 53. As with rule statements, post evaluation checks used the structure editor to update their expressions. See Figure 54 for a function editor with a single post execution constraint specified.

```
MIN_CONSTRAINT: <numeric expression>
MAX_CONSTRAINT: <numeric expression>
MIN_ERROR: <numeric expression>
MAX_ERROR: <numeric expression>
```

Figure 53. Incomplete post evaluation checks.

A last feature of the function editor was the incorporation of diagnostics. The language did not allow for user-defined print statements within functions and, given how much the users had relied on diagnostics in the former language, I decided I needed to provide them with a means of debugging their rules and functions. To do this, I added the ability to automatically display information about the execution of a function both before and after it executed. Before execution, the function could print out its name and the values for each of its arguments. After its execution, it would print out its return value.

One issue related to the diagnostics that needed resolution, however, was the scale and units that would be used to display the function's return value. It would do little good to display the return value in the default scale and units since that would require the users to convert them to the scale and units that they would normally use. Instead, I included a text field to allow the user to specify the scale and units in which to display the function's return value. As with the rest of the rule language, unit checking would be performed at run-time. If the value could not be converted to the specified units, an error would be generated and the simulation terminated. Although I wanted to and felt it necessary, I did not have time to provide scale and unit specification for the function's arguments. This would have to wait until a more structured approach to specifying arguments was incorporated into the function editor.

For an example of a function editor with diagnostics turned on and units specified, see Figure 54. Since a scale is not specified, it defaults to 1.0

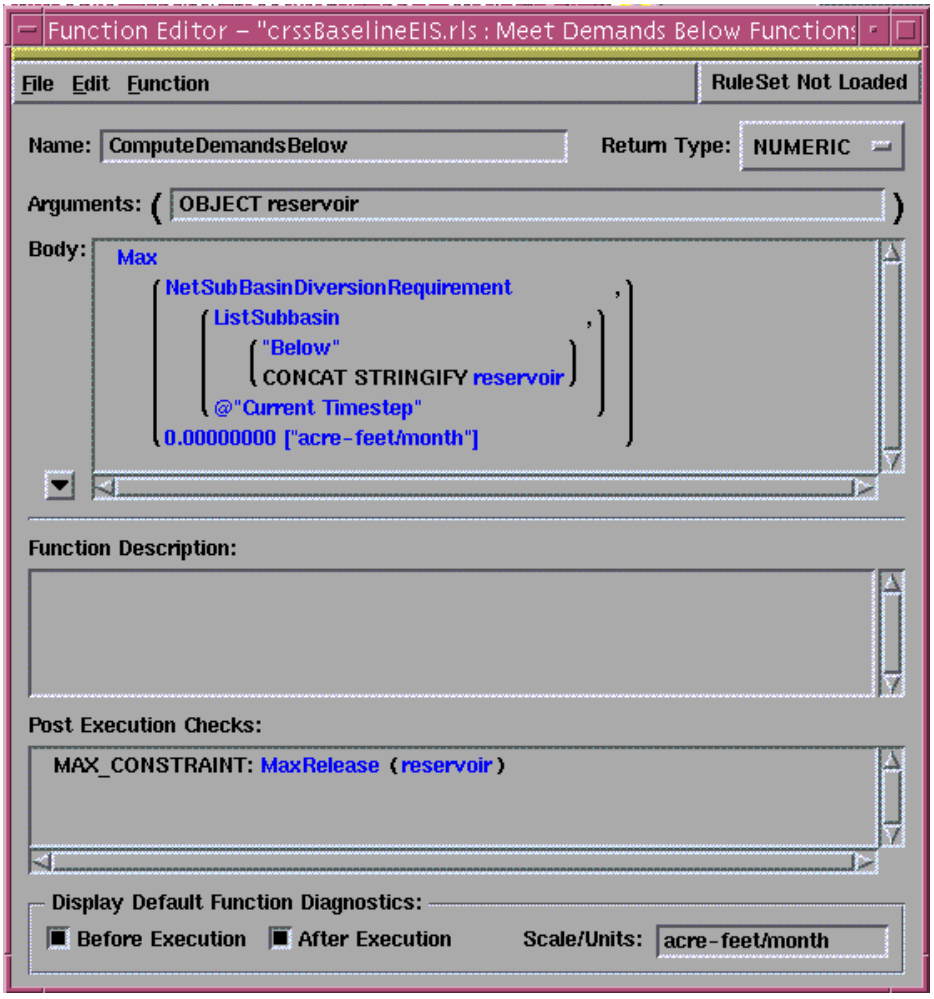

Figure 54. Function editor with a post evaluation constraint and diagnostic information specified.

5.7.3. Issues

5.7.3.1. Structure Editors and Experienced Users

Although I felt that structure editors would be useful in addressing a number of the problems associated with the creation, maintenance and reading of code by inexperienced programmers, there was one drawback to their use. In general, structure editors are easy to use and useful for new programmers, but can become cumbersome to use as users gain experience. This is largely due to the fact that structure editors control the content of the programs to insure that they are always syntactically

valid. Thus, they prohibit the creation, even temporarily, of programs that are not valid. While this prohibition can be useful in insuring the users' programs are never syntactically incorrect, they can force users to create their expressions in a manner that may not be to their liking.

5.7.3.1.1. Example

Figures 55 and 56 contain series of `IF` expressions that show two different editing progressions. The first expression in each figure shows the initial state and each subsequent expression shows how the expression changes during editing. Note that "?"s are used for placeholders so as not to restrict the manner in which expressions are added to the language.

In the example in Figure 55, the user substitutes the placeholder in the antecedent with an expression that consists of two placeholders and the ">" operator, which keeps the entire expression syntactically valid. In the next two substitutions, the two new placeholders are replaced with object.slot references and the antecedent is complete and syntactically valid. In fact, it has been syntactically valid the entire time, although it has not always been complete.

```
IF ( ? ) THEN
   ?
ENDIF

IF ( ? > ? ) THEN
   ?
ENDIF

IF ( mead.outflow[] > ? ) THEN
   ?
ENDIF

IF ( mead.outflow[] > mohave.outflow[] ) THEN
   ?
ENDIF
```

Figure 55. Valid editing sequence.

In the example in Figure 56, however, the antecedent is replaced with a numeric expression. This is not syntactically valid, since a numeric expression cannot be the antecedent of a conditional expression. This is true even though a subsequent substitution might be used to replace the numeric

expression with a relational expression in which the numeric expression now occupies the left operand. So even though the resulting expressions are the same, the structure editor would not allow the second sequence because it would cause the entire expression to be syntactically invalid at one of the steps.

```
IF ( ? ) THEN
   ?
ENDIF

IF ( mead.outflow[] ) THEN            // INVALID
   ?
ENDIF

IF ( mead.outflow[] > ? ) THEN
   ?
ENDIF

IF ( mead.outflow[] > mohave.outflow[] ) THEN
   ?
ENDIF
```

Figure 56. Invalid editing sequence.

5.7.3.2. Solution

This restriction on the order of creation was likely to be a problem given that the original USBR users had gained sufficient programming experience that they could find the creation and editing of their rules to be too slow with a structure editor. This motivated me to see if I could satisfy both sets of users. An obvious solution to this problem was to provide a readable text-based base language. The structure editor would use this base language to format and display the programs written using it. It would also use this language when it saved the programs to file. A traditional text editor could be used to edit the underlying language directly. While there was some risk associated with allowing the users to use the base language directly, this was no more than would be found when using a text-based language without a structure editor. This solution allowed both sets of users to determine which means of editing they preferred.

5.7.3.3. Multiple Rulesets

Users had expressed an interest in simultaneously editing multiple rulesets. Moreover, if they wished to transfer language elements between rulesets using the programming environment, they

needed to have both rulesets open. Given my experience during early development and unit testing, I came to the conclusion that it would be confusing for the users to have multiple rulesets open at one time without providing some means of telling them apart. At first I thought it would suffice to put the name of the ruleset in the window manager's title bar. Unfortunately, this was difficult to see, especially with many editors open. Next, I tried color coding the background for all the editors based on their ruleset, which resulted in one ruleset having editors with blue backgrounds and another having editors with green backgrounds. While this worked to differentiate the editors for each open ruleset, it also made the screen difficult to look at given all the colors. Lastly, I settled on colored trim across the top of each editor. This trim amounted to a thick line that could be seen yet did not degrade the appearance of the entire screen.

I also decided that I could use this colored trim for an additional purpose. I wanted the users to know when a ruleset was loaded into the simulation engine so that they would realize that their changes would affect the next simulation. To do this, I changed the trim color to red for all of the loaded ruleset's editors. In addition, a button located just under the colored trim changed wording from "Ruleset Not Loaded" to "Ruleset Loaded." Figure 57 contains an example with editors from three rulesets, one of which is currently loaded.

5.7.3.4. Screen Space

Another issue related to the previous one is how best to manage scarce screen real estate. Given that there could be quite a few editors open at any given time and that *RiverWare* itself could have a number of windows open, I decided that it would be a good idea to try to keep the editors small. Hudson discusses a technique of showing important information and hiding less important information by default while still allowing the less important information to be displayed if desired [Hudson, 1994]. I found this to be an appealing idea and incorporated it into the design of the programming environment editors. This resulted in each editor having a relatively small main screen in which the primarily accessed information resided. There was also a means of extending this window to display other information, that did not always have to be viewed. In this way, users could view information depending

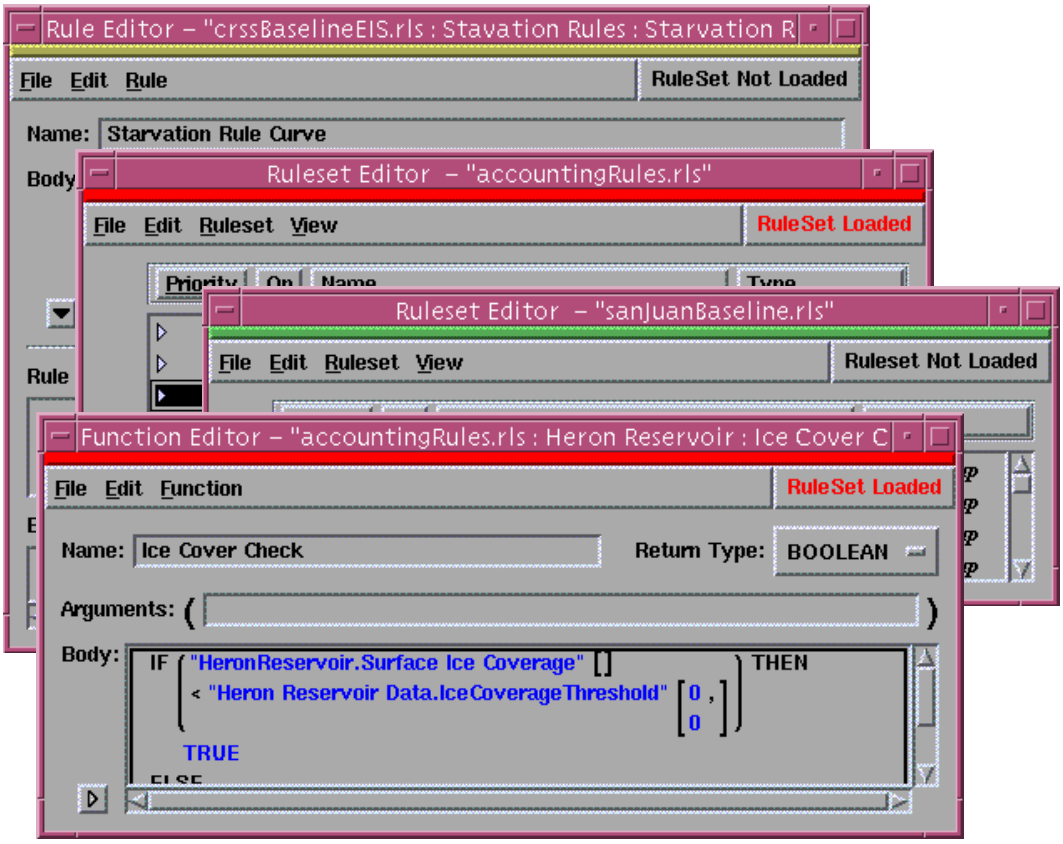

Figure 57. Editors from multiple rulesets.

on their needs and not be required to take up screen space that could be better used by other dialogs. There was risk involved with this approach in that it would be possible to miss information that was on the hidden part of the screen. Yet, I felt the benefits of being able to display more editors outweighed the drawbacks. Figure 43 show a function editor with only the main portion of the display visible and Figure 54 shows a function editor with all information visible.

5.7.4. Ruleset Validity Checking and Loading

Before a ruleset could be used by the *RiverWare* simulation engine, it needed to be both valid and loaded. While the structure editors ensured that the ruleset was syntactically correct, they did not require the rules and functions to be fully specified. In addition, object.slot references, which were required to be valid when entered, could be rendered invalid by subsequent changes to a *RiverWare* model. As a result, a ruleset could be incomplete and/or incorrect without users being aware of it. To

help users determine the completeness and correctness of their rulesets, the editors provided checks that could be invoked on an editor-by-editor basis. These checks would analyze the requested part of the ruleset and all of it children to insure that the elements were ready to be executed. Any problems would be reported to the user.

In addition to validity checking, the environment also needed to provide a means of loading a ruleset for use by the simulation engine. To do this, users were required to select a ruleset file using a file chooser. A validity check was then performed on the selected file. If any errors were found, users would be alerted and the ruleset load would be terminated. If no errors were found, the ruleset would be marked as loaded and its open ruleset editors updated to show users that this ruleset was loaded and that any changes would be reflected in the next simulation execution.

5.7.5. Abnormal Execution Feedback

One last feature that was included in the initial language and that the users found useful was abnormal execution feedback. This was necessary to show users the nature of any errors that might occur during a simulation execution. Examples of errors included division by zero, external functions returning a value with an invalid type and the use of incompatible units in relational and arithmetic expressions. Once an error was found, the user would be provided with a trace of the error, a brief explanation of the problem and a reference to the offending expression.

5.8. User Tests

Two user tests were performed for this case study, one in October 1997 and the other in February 1998. They were both on smaller groups than were used for the first case study's user test and were part of a CADSWES-sponsored training course. In addition, because the language and programming environment were so new and had not been used much, the first user test was preceded by a 'friendly' pre-user test to work out as many obvious usability problems and bugs as possible. This would allow me to gain an understanding of the type of problems that the users might encounter and

structure the training accordingly. This chapter will contain an overview of the user tests, while the bulk of the ‘lessons learned’ are presented in the next chapter.

5.8.1. Pre-User Test

In order to properly prepare for this training course, I requested and received the specifications for the problem that the target users were going to work on. It was not written in a way that matched the rule language. My intent was to prepare a model and ruleset that corresponded to their system and test out the model, rule language and environment to insure that users could effectively work on their problem. In addition, I wanted to identify and have time to fix any glaring problems that users would otherwise encounter during this process. Lastly, I wanted to try to gain an early understanding of some of the problems that users might have with the new language and its environment.

I started the process of translating the rules with one of *RiverWare’s* designers. She was familiar with the original rule language and knew about the new rule language at a conceptual level. Initially, she found the functional language difficult to use and the translation from the procedural representation of the rules nonintuitive. This seemed primarily due to a lack of understanding of how the new language worked and, after a brief explanation, found the transition from the first language relatively easy to make. The initial difficulty lay mostly in the fact that the original language allowed for the setting of multiple values within a single rule while the new language required a distinct rule statement for each value to be set.

After that, I moved to a user with a great deal of experience using the previous language and who was somewhat familiar with the new language. He had no documentation or training and his initial impression of the language and programming environment was negative. After some use, however, he grew to like the new language and environment more than the original. This was significant, given that he was the original language’s most enthusiastic supporter. His use of the language proved very helpful in that he was able to identify a number of usability issues and bugs that had either not been found or not recognized. We were able to fix most of them before the main user test.

5.8.2. First User Test

The user testing took place as part of a three day *RiverWare* training course. This training course was specific to the rulebased simulation component of *RiverWare*, although by necessity touched on many of the general aspects of *RiverWare* as well. The users' goals were to learn how rulebased simulation worked, learn about the rule language and environment and how to use it, and to apply what they were to learn to a particular problem.

Four users were involved. All were civil engineers and all were involved in river basin modeling as part of their jobs. None of them had any experience using *RiverWare* or its rule language. All had some experience with procedural programming languages, although programming was, at best, a secondary part of their job.

5.8.2.1. First Day

On the first day, the users learned about *RiverWare* and its capabilities. This included instruction on its GUI, its method of doing mass-balance simulations and its means of specifying inputs and viewing outputs. In addition, the users went through a simple rulebased simulation tutorial. The model was a simple two reservoir model and the ruleset was a simple three rule model. This model and ruleset combination was similar to the one that was discussed in Section 2.4.3.2. Both the models and the rules had been previously created to allow the users to focus on how they interacted with each other rather than their construction.

In order to allow the users to understand the new rule language and how the simulation used rules, they were asked to open and view the rules, but not make changes to them. Instead, they were asked to run simulations using the rules as given to them and to 'turn off' (i.e., deactivate) rules and re-run the simulation. After each of these simulation runs, they were instructed to view the output generated by the simulation to see the effect of the rules on the simulation. I did not participate in this part of the user training.

5.8.2.2. Training

The second day began with an overview of the rule language and how it was used by *RiverWare*. Since the users already had some exposure to this from the previous day, the overview primarily centered around the functional nature of the language and how it related to the more traditional, procedural languages with which they were familiar. The topics included the structure of the language with its rulesets, groups, rules and functions; the language's types; the language's statements and expressions; and examples of how to use these to create some simple rules. This session lasted just over an hour and did not go into great depth.

Following this session, a second session was convened that used one of their target policy statements as an example. As mentioned previously, the users had sent us the policy statements that they wished to write using the rule language. The intent of this session was to walk them through an example that they were familiar with and to show how they would create one or more rules for this policy statement. Since the form in which the users had sent the policy statement was procedural, we started from this and worked out how to transform this into an equivalent set of rules in the new language.

The conversion process went smoothly, although it was clear that the shift from a procedural to a functional paradigm was not an easy one for them. One user felt that a flowchart would be easier to use when explaining a policy to a manager. Another felt that he would have to write his policies procedurally and then convert them into the functional form required by the new language. Given that the language was created to avoid both of these, I was concerned about the outcome of this training session.

After this session was finished, I got some mixed comments. The users seemed cautiously optimistic, although some expressed some uncertainty as to how well they thought they could use the language. Given that the comments were made in my presence and that they knew I was the language designer and implementer, I took the positive comments lightly and the uncertain ones more seriously.

5.8.2.2.1. Hands on Use

The rest of the second and the entire third day were spent having the users create the ruleset that we had discussed in the previous session. I informed them that I would give them as little help as I thought was reasonable for them to create their rules. I did this so that I could observe what seemed intuitive and what was difficult to understand. I also asked them to 'think out loud' while they worked so that they would vocalize any questions or comments they had while working [Lewis, 1982]. On the second day, the users were split into groups of two. I observed one group while the other group spent time working on a more detailed design of the entire policy for the ruleset. On the third day, all four users worked together.

By and large, this part of the user test/training went well, although there were a number of bugs found that proved to be annoying. Both groups were able to create and successfully load their rulesets, although not without some difficulty. While some of this difficulty can be traced to the bugs and others to design flaws, others can be traced to having users perform tasks on a system for which they had little training and no documentation. While increasing both of these would have certainly made use of the tool easier, it would not have yielded as many insights into the users' experiences with the language and programming environment.

5.8.3. Second User Test

The second user test took place in February 1998 and included three users. This user test was similar to the previous one in that it took place in the context of a user training session in which users came with their own problem to solve and spent time studying their problem, seeing how it fit into the new language and then using the new language and programming environment.

As before, the user training took three days. The morning of the first day was spent discussing the language and how it differed from a traditional procedural language. Since two of the users had used the previous Tcl-based rule language and the other user had used FORTRAN for about twenty years, they were familiar with procedural programming. With the exception of one of the users (UserA) who

had spent some time on his own using the new language and programming environment, the users did not have any significant experience with a functional language. The afternoon of the first day and the entire second day were spent discussing their problem and how they would express it using the new language.

The third day was spent creating their rules using the system with the three users working together. The set of rules and functions was only a subset of what they needed to express. In part this was because they wanted to learn the language and environment and in part this was because some of their problems required extensive list manipulation capabilities, which had not built into the language yet. Having this group work together had an interesting result. Unlike the first user test where the users had no rulebased simulation experience, the second group had only one user (UserB) without rulebased simulation experience. One user (UserC) had used the previous language and environment for about a year and the last user (UserA, the same as in the pre-User Test) had had extensive experience with the old language and environment and had used the new language and environment on his own for a number of months. As a result, this group of users was quite a bit more advanced and knew what they wanted both from the language and from *RiverWare*. In addition, since they were working together they were able to help each other and progress more quickly in creating their rules.

Since these users were more advanced, I started them out with an introductory demonstration. I showed them how to create rulesets, groups, rules and functions and how to use the environment. Since I had already gotten feedback on the use of the system without any initial help, I wanted to let the users get a good initial understanding and then see what problems they had. In general, this user test went far better. This was in part because I had given them the initial demonstration and because I had had time to fix most of the bugs that had been discovered during the first user test. It was also because the users, as a group, had more experience with both *RiverWare* and rulebased simulation.

Chapter 6

Case Study 2 Lessons

6.1. Introduction

The language and programming environment developed for the second case study were considered by the users to be superior to those developed for the first. They met their needs in terms of expressiveness and, as importantly, in terms of readability. Many of the characteristics of the new language and environment were a direct result of what was learned during the design, development and use of the first system and it would have been difficult to have designed the second system had the first one not been created and its use observed. While creating a language and environment, learning from them and then creating a subsequent language and environment might be acceptable in the context of research, it may prove to be unacceptable in situations where development time and cost need to be minimized. To that end, this chapter will discuss some of the lessons that were learned as a result of the design, development and use of the language and environment in hopes that they will shed further light on the issues that may be encountered during the design and development of an application-specific language system. The lessons will be grouped by the language, the programming environment and the design methodology.

6.2. Language

The second programming language was quite different from the first. It was a task-specific and functional language rather than general purpose and procedural and, as a result, it put more constraints on how the users could represent rules and functions. These restrictions did not impede the expressiveness of the language; the users were able to duplicate their original ruleset using the new language. The restrictions did, however, alter the required representation of the rules and functions in the ruleset. These restrictions, coupled with the visual conventions imposed by the programming

environment, resulted in a language that better met the users' needs with regard to readability, which was a significant failure of the first language. This section will discuss some of the issues that surrounded the design and development of the second rule language. These will include the language type, its characteristics, the goals that helped to lead to the language and some of the issues surrounding the use of the language.

6.2.1. Type

6.2.1.1. Functional

As discussed in the previous chapter, the use of a primarily functional language was largely an outgrowth of my attempt to break down the Tcl-based policies created in the original language so that I could understand the entire policy by understanding the meaning of each of its parts. While I had not set out to create a functional language, it became clear that this form would go a long way toward meeting the goals of readability and maintainability that had not been adequately met in the previous language. This was largely due to the fact that the new language constrained the representation of the users' policies to a set of side-effect free functions that could be understood outside of the context of their execution. While the previous language permitted a similar representation of the users' policies, it did not explicitly require it. This, coupled with the inexperience of the users, resulted in rules and functions that were often monolithic and difficult to read. Lastly, the requirement that all assignments be performed in the rule made it clear which object.slots could be set by a rule and under what conditions they could be set. This was in contrast to the previous language in which object.slot values could be set anywhere, which could result in searching all the code in a rule and its functions in order to know this same information.

While the use of a primarily functional language had some significant advantages, it also had some drawbacks. The main one was the paradigm shift that was required of the users. Most of the users had at least some programming experience, although it was invariably using procedural languages. As a result, they often found it initially difficult to use the language and struggled to translate their policies into it. In addition, they were resistant to the idea of using recursion, which will be discussed in more

detail in Section 6.2.2.1. In general, however, after some hands-on instruction and some time spent using the language, the users generally became comfortable and productive with it. The environment more than likely played a significant role in this and had the users been presented with an environment-free version of the language, they would have had more difficulty learning and using it.

On the whole, the selection of a primarily functional language was positive. It allowed me to constrain the form of the rules and functions such that the users created smaller sections that could be more easily understood and combined than the larger, more general rules and functions allowed in the previous language. While it initially made it harder for the users to write their rules, they quickly overcame this. In fact, given that the users' policies consisted primarily of logical and arithmetic expressions, which relied on the use of functions, object.slot look-ups and literals, the use of a functional language fit nicely.

6.2.1.2. Visual

I had originally assumed that I would develop a visual language to replace or overlay the original Tcl-based language, which was considered a short-term solution. I felt that a visual language would be easier for inexperienced users to use and that I could come up with a visual representation of the policies that would be superior to any textual representation. After some research into visual languages [Petre, 1995, Green and Petre, 1992, Petre and Green, 1990, Petre and Price, 1990] and some design attempts, however, it became clear that this was not so. The policies were primarily logical and arithmetic expressions that did not lend themselves to a visual representation as much as they did to a formatted textual representation.

The lesson here is that the problem descriptions and user goals should drive the language design rather than some preconceived notion of what the final language should be like. Given that *RiverWare* models had some visual programming characteristics, it seemed natural to have a policy language that interacted with the objects in models also make use of visual programming. Yet, the reason that *RiverWare* models could successfully use these visual elements was that their elements

represented the physical layout of objects on a river basin, which was by nature spatial. Accordingly, the relative locations of reservoirs and other water objects was relevant. This kind of spatial representation was not as applicable for policies, which were concerned with the values that the objects stored rather than their spatial relationship. In addition, even though *RiverWare* models were represented visually, the underlying characteristics of the objects were not. Numbers, for instance, were displayed textually and directly editable by users. In contrast, a precursor to *RiverWare* had required numbers to be edited using sliders, which the users found to be cumbersome and inaccurate.

6.2.2. Characteristics

There were a number of language characteristics that affected the users' experience with the language or were important in the design process. These included its control structures, its types, the placement of its comments and the use of a single ruleset file.

6.2.2.1. Control Structures

I spent a considerable amount of time working on control structures that would avoid recursion. Users had asked that the language not require the use of recursion and, given their negative experiences with it in the *RSS* rule language, I took this request seriously. In order to provide the users with the ability to express their policies in a manner that was familiar to them, I took imperative control structures and altered them in order to make them fit within the context of a functional language. Two control structures, a `FOR` loop and a `WHILE` loop, were created that allowed the users to use a limited and self-contained form of iteration within rules and functions.

While the control structures worked as intended, they were not as easy to use as I had hoped. For simple value computations, the control structures were difficult to write and subsequently understand. In part, this can be attributed to the fact that their syntax did not make it obvious what they did. To address this, the use of a programming environment-based template that made it clear what inputs were needed, what value was being updated and how, and what output would be produced would likely have been helpful. The underlying language syntax could have remained the same. For example,

the FOR loop could have been changed from the example in Figure 36 on page 150 to include an explicit initialization, assignment and return value as in Figure 58. The template could make sure that the name used on the initialization line, on the left-hand side of the equals sign and on the return line were always the same. While this syntax does not strictly conform to the rest of the language, it does more closely match the imperative syntax that the users would likely understand and, as a result, makes it more obvious what the FOR loop is doing. Similarly, the WHILE loop could have been changed from the example in Figure 38 on page 151 to that in Figure 59.

```
INITIALIZE result = 0
FOR ( NUMERIC index IN {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} )
DO
   IF (index MOD 2 != 0) THEN
      result = result + index
   ENDIF
ENDDO
RETURN result
```

Figure 58. Alternative FOR loop.

```
INITIALIZE powellOutflow = 0
WHILE ( ABS (PowellStorage (powellOutflow) - MeadStorage (powellOutflow))
        > delta )
DO
   powellOutflow = powellOutflow + increment
ENDDO
RETURN powellOutflow
```

Figure 59. Alternative WHILE loop.

Yet, even if these syntactic conventions had been adopted to make the simple cases more understandable, it is unclear that for more complex logic these control structures would have been any easier to use or understand than recursion. This came to light while I was trying to verify that the control structures could be used to represent one of the users' policies. In *Reservoir Equalization*, the storages for two adjoining reservoirs (Lake Mead and Lake Powell) are required to be approximately equal. To insure this, the policy iterates over their storage values, while manipulating their inflows and outflows and the inflows and outflows for surrounding reservoirs, until the storages of Lake Mead and Lake Powell are within some predefined tolerance. Although the calculations needed to compute the storages

are fairly complex and interrelated, the basic structure of this rule can be fairly easily represented using a traditional *while* loop.

I was able to represent this policy using the new `WHILE` loop, although it took some time and the resulting rule was not particularly easy to understand. Given this, it is likely that less experienced programmers would also have trouble representing this policy and that non-programmers would have trouble even understanding it. As a result, these control structures do not meet the goals of the language. Unfortunately, however, none of the available alternatives seemed to offer an obviously better solution. The use of recursion was unlikely to be embraced and, even if it were, would not necessarily provide the users with a solution in which they would easily create and maintain their rules and functions. In addition, it is unclear that non-programming users could understand rules or functions that made use of recursion. The use of external functions would allow some of the users to produce solutions that required either iteration or recursion, although most would not be able to do so and even fewer would be able to understand the solution. To a large number of users, these functions would become black box functions, which would allow them to neither experiment with their underlying policies nor understand their intricacies. Other possible solutions would be the creation of different, perhaps better, control structures, the ability to use local variables within functions or further enhancements to the programming environment. In hindsight, it is possible that user tests that focused on the three available alternatives – the new control structures, external functions and recursion – might have yielded some insights into which approach best met users' needs.

6.2.2.2. Types

The final language had eight data types, although I initially anticipated only four. The four I anticipated, numeric, boolean, string and lists, were later augmented by object, object.slot, date/time and unknown. I believe the initial omission was primarily due to the fact that I was thinking in terms of a general purpose language rather than an application-specific language in which the host application's types would likely need to be supported. Although no permanent damage was done in initially omitting these types, adding them later was more difficult than if their need had been initially recognized. The

only type that was not obvious based on the host application was the unknown type. This type came to light because of the use of the structure editor and its need to save and load incomplete, but syntactically correct, expressions.

6.2.2.3. Comments

Comments were included in the new language, but their number and location were tightly constrained. This was in contrast to the first language in which the placement of comments was left to the user. While this might strike some as a step backwards, the restrictions were largely due to how the language elements would be used by the structure editors. In most languages, comments are placed in the code using a free-form text editor and stripped out prior to the code's compilation or execution. In the new policy language, everything that was part of the ruleset needed to be loaded, displayed, edited and saved using the structure editor. If the language had contained arbitrarily placed comments, this would have entailed considerable design and development time and it is unclear that the resulting language would be more readable.

One place where extra comments should have been added was in rules. As was discussed in the previous chapter, a comment statement could have been added to the rules. This would have taken little extra development effort and would have allowed the users to have one comment associated with each assignment and conditional statement.

6.2.2.4. Single Ruleset File

The use of a single ruleset file was a departure from the previous language in which the users were required to create and maintain multiple UNIX files. There were two main motivations for using a single file. The first was to allow the users to be almost entirely insulated from the UNIX environment while they were working with their rulesets. Their only interaction with UNIX would be the file selection required to load or save a ruleset file. The second was to allow a ruleset to be stored in a single location, which would facilitate any administration and maintenance that might need to be performed

outside of *RiverWare*. This would generally not be done by those who did not have some experience with UNIX, but for those who did, it would make these tasks much easier.

The main drawback to having a single ruleset file was that it made sharing portions of a ruleset difficult. To offset this, the programming environment included functionality that allowed users to move and copy rules and functions between rulesets. The only capability that was not supported, either in the language or in the programming environment, was sharing rules or functions by reference. One example method of providing this functionality would be to have the main ruleset file reference one or more ancillary ruleset files that would contain the rules and/or functions to be shared. This would have provided much of the flexibility that was lost when I switched to one file, although it would have required the use of more than one file, which was what I was trying to get away from. While I had considered adding this functionality, I did not have time and felt that until it was explicitly requested there were other, more pressing needs, to be addressed first.

6.2.3. Goals

This section discusses some of the goals of the language and how they were or were not met. Only safety, performance, familiarity and backwards compatibility will be considered here as the other, relevant goals have been or will be covered in other sections.

6.2.3.1. Safety

One of the primary goals for this design effort was to make the second language safer than the first. As was discussed in Chapter 4, the first language had been used in ways in which it was not intended. This was largely due to the fact that the language permitted this misuse and the users were often more interested in getting their rules to produce the results that they expected than using the language in the manner in which it was intended. Perhaps this should have been expected and accounted for. In the first place, the users could not necessarily have known the manner in which the language was intended to be used; they were novice programmers who performed programming as a secondary part of their job. In the second place, even when they did know, their goal of getting a working ruleset generally

outweighed their desire to follow the programming guidelines that they were told were important. This was particularly an issue given that the users were not experienced programmers and, as a result, had had little exposure to the consequences of poor programming practices.

There were three notable characteristics of the new language that made it a safer language when compared to the first language: the removal of global variables, the addition of automatic unit checking and conversion, and the addition of automatic dependency determination. Each of these three characteristics had been misused in the previous language. Global variables had been used to create functions with side effects, which had made it easy to inadvertently break rules by making seemingly innocuous changes. The use of functions with side-effects had also impaired both readability and maintainability. Values with incompatible units had been incorrectly combined and compared, which led to subtle and potentially undetected errors. Dependencies had been both intentionally and unintentionally incorrectly specified, which led to poor performance in the best case and incorrect simulation results in the worst.

While each of these three changes improved the safety of the language, their inclusion was not without cost. Removal of global variables slowed the initial creation of the rules. Unit checking and conversion increased the design and development time and slowed the execution of the rules. Automatic specification of dependencies increased the design and development time and slightly increased the execution time of a rule. Yet even given these costs, it is difficult to argue that these safety features should not have been included in the language. Policies created or executed quickly are of little benefit if the results they produce are unreliable. In the best case, these results could cause the rule writers to spend extra time debugging and correcting their rules. In the worst case, the incorrect results could lead to the policies being put into practice on a real river basin, which could lead to a misallocation of water or even flooding.

The main lesson here is that a language that is fast but that produces potentially incorrect results is unlikely to be the correct approach when dealing with inexperienced programmers. One might argue that this is not even a good idea when dealing with experienced programmers. While there are

undoubtedly situations in which it can be argued that performance requirements dictate a speedy and unsafe language, one has only to look at the Mars Climate Orbiter that crashed in September 1999 due to a failure to convert English units to metric units to realize that language safety is important and the cost of ignoring it can be tremendous [Mars, 1999].

6.2.3.2. Performance

I initially began the design of the language with full confidence that I could create one that was faster than the Tcl-based language. A language that executed the rules quickly was important because of the complexity and size of the simulations. In the worst case – modeling the entire Colorado River Basin over eighty years – these simulations could take upwards of eight hours. Given that the Tcl-based language was not as fast as the users wanted it to be and that I needed to add features such as unit checking and conversion and automatic dependency determination, I felt that a custom language would be the best option for providing both the performance gains and the needed safety features. While the language was able to eventually match the speed of the Tcl-based language, this was far worse than I had hoped for.

In part, the language's poor performance was a result of the added features just mentioned. A significant part, however, was due to the fact that designing an expressive language that can be used to create complex rules and functions was a difficult task and inexperienced language developers were unlikely to be able to create a language in six months that could equal the speed of mature languages, such as Perl or Tcl, that were developed and are maintained by a team of accomplished language developers.

6.2.3.3. Familiarity

People tend to prefer what they know and this should be taken into account when designing a language and environment for a specific group of users. It is likely that the language being developed will be used to replace an existing method of solving users' problems. This method, which is probably imperfect, is at least known to them and they may be comfortable using it, despite any problems it might

have. As a result, it is important to realize that users may not quickly embrace the new solution. As an example of this, a few primary users of the first language were hesitant to embark on the design and development of a new language. They could use the first language to get their work done and there was no guarantee that the new language would be any better. In addition, they would have to learn a new language and convert all of their existing policies. While I think they understood the need to replace the first language, the cost of doing so seemed high.

In order to address their concerns, I used a few different approaches. One was to involve some of the users in the design of the new language. I did this by keeping them informed as the design progressed and involving them in walkthroughs using their problem descriptions once the design was far enough along to give them a good idea of what their rules would look like and how the structure editor would work. This approach was helpful, although it had a potential drawback. Since the users were involved early, subsequent design changes could have led to a lack of confidence or confusion on their part. Fortunately, this was not the case, although open communication about the design process and the motivation for any design changes probably helped. Had I not involved the users until later, there was a chance the design would not have met their needs, since I may not have received feedback in time to easily make any necessary changes. In addition to feedback on the design, the involvement of users can have another benefit. It can serve to gain the users' support for the language and environment, since they are part of the design process and may well assume part ownership of the finished product.

Another approach I used, which will be discussed in more detail in the following section, was to allow the users to create functions using the old language from within the new language. This gave the users the confidence that they could always fall back on the old language if the new language didn't meet their needs. As it turned out, it also provided a method of allowing the users to slowly transition their policies from the old language into the new.

6.2.3.4. Backwards Compatibility

The inclusion of external functions in the second language turned out to be one of the surprise successes for the language. I had envisioned external functions as an infrequently used backdoor into the first language to be used only when the syntax of the new language lacked the needed expressiveness. While they can be used for this reason, the primary reason for their use, at least initially, was to provide a means of transitioning the rules written in the previous language to the new language. In hindsight, this makes sense given the amount of time that the users had spent constructing and verifying the correctness of their rulesets written using the first language and their hesitancy to start from scratch with a new language. The use of external functions allowed them to slowly transition from the previous language to new language while verifying that each step of the conversion produced the expected results. One notable example of this that pleased the users was during their initial attempts at converting the entire Tcl-based ruleset into the new language using mostly external functions. They managed to write all of the rules, create and convert all of the high-level Tcl procedures into external functions, and verify the results in three hours. From here, they were able to slowly transition the contents of the external functions into internal functions, verifying the results as they went.

The lesson here is that it is important to provide a means of transitioning the programs written in a language to the next generation of that language. For commonly used languages this is unlikely to be an issue since a new language is almost invariably backwardly compatible. With an application-specific language, however, it is possible that fundamental changes will be made to the language that will make backward compatibility difficult. Yet, even if the language is fundamentally different, as was the case with the two *RiverWare* policy languages, it is important that the users not be required to immediately rewrite all of their programs in order to make use of the new language. While this seems obvious, I did not explicitly plan for it. Instead, I was fortunate and added the functionality necessary to support backward compatibility for another reason.

6.2.4. Use

This section will discuss some of the areas related to the use of the language. These include the language's error messages, its debugging facilities and the users' ability to directly edit rulesets.

6.2.4.1. Error Messages

As with most languages, the *RiverWare* policy language generated run-time errors. And because the language was to be used by inexperienced programmers, these run-time errors would be read and interpreted by inexperienced programmers. As a result, it was important that they be as clear and non-technical as possible. In this, the language was only partially successful. While attempts were made to keep the wording of the errors as accessible as possible, there were instances in which difficult to use wording was used. In addition, even when clear wording was used, the source and location of errors was given in the context of a purely textual and often lengthy traceback message. While traceback messages were helpful, they could also be difficult to decipher.

An example of a fairly opaque error message was "List subexpression of FOR expression didn't evaluate to a list (it evaluated to a value of type NUMERIC)." This error message would be accompanied by a traceback of the error and a textual representation of the `FOR` expression that contained the error. While this presentation might be helpful to an experienced programmer, it could prove to be unclear to an inexperienced programmer who might not understand what a "List subexpression" was and why it mattered that it "didn't evaluate to a list." Perhaps more importantly, the traceback message was in a textual form and did not contain the same formatting that the user would have seen in the context of the structure editor. This would make it difficult to read, especially if complex subexpressions were used within the `FOR` expression.

A better approach would have been to display the error message in the context of a structure editor, in which the portion of the expression that contained the error was highlighted. In the above example, the portion of the `FOR` expression that was supposed to be a list would be highlighted. This would need to be accompanied by a text-based explanation as well, since just identifying the location of

the error would be unlikely to provide enough detail to permit the user to address the error. At a minimum, this would make it clear to the user where the error occurred and what the error was. An even better solution would be to allow the user to step back through the call sequence to determine the exact nature of the error. This is the approach that is used to great effect by the Smalltalk programming language and environment [Lewis, 1995].

6.2.4.2. Debugging

The original language had included trace statements that printed out user-specified information as rules and functions were executed. They were extensively used to debug the users' rules and functions. The new language, however, was more structured and, as was discussed in Section 5.6.2.2, alternatives to free-form trace statements were needed. While these alternatives constrained the manner in which users could display debug statements, they worked well. There were two main areas, however, where improvements should have been made. Both related to the trace statements used by functions.

Recall that functions allowed users to display the values of their inputs before they were executed. One capability that was not permitted was the ability to print out other values used by the function, such as object.slot values. This omission could make it difficult for users to understand how a function calculated its result if object.slot values were used in the calculation. The other capability that was lacking was the ability to specify the scale and units of the function's input parameters. This meant that each of these values would be displayed in standard (i.e., metric) units and would possibly require the users to convert them to the scale and units to which they were accustomed. This omission was especially notable given that functions did allow the users to specify scale and units for the trace statement on the function's return value. The former issue resulted from not being aware of the problem until this writing; the latter was omitted due to a lack of development time. While both of these problems would have taken some time to resolve, their inclusion would substantially improve the users' ability to debug their rules.

6.2.4.3. Accessible Text Language

In addition to the structure editors, I wanted to provide the users with the ability to edit their rulesets directly. This was meant to allow advanced users to avoid the structured editing environment if they chose to and to allow anyone to make changes to rulesets if they were not currently running *RiverWare*. This latter case would be necessary when the users were running the non-GUI version of *RiverWare* and needed to update their ruleset. In order to allow the editing of *RiverWare* rulesets outside of the context of the structure editors, the underlying, textual language needed to be as humanly readable as possible. This meant that I could not store a ruleset in binary form and that the language should not contain extraneous formatting elements, such as markup tags. While users could edit rulesets if they contained markup tags, it is likely that they would have found it more difficult than without them.

One practical consequence of this decision was that there was an increased possibility that the ruleset file could be corrupted by the users. Yet, even if only the structure editor wrote the ruleset file, there would always be a chance of some kind of corruption. In addition, there was no guarantee that users would not inadvertently try to load a file that was not a ruleset file. As a result, the contents of each ruleset file would need to be verified as it was loaded. None of these constraints were unreasonable or difficult to implement and the language could be used either directly or through the structure editor. On the whole, this was a positive characteristic of the language and one that I would recommend in cases where a structured editing environment is included.

6.3. Environment

The programming environment for the second language was a departure from that provided for the first language. There was more programming support and many of the glaring problems, such as the inability to save a ruleset, had been corrected. In addition, the environment provided the users with a better visual representation of their rules and functions and provided a great deal of editing support in the form of a structure editor. Given the differences, it is not surprising that there were many new problems encountered and solutions required. In addition, there were new lessons learned, which will be discussed in this section.

6.3.1. Structure Editors

The programming environment relied heavily on the use of structure editors that constrained the manner in which expressions were viewed and edited. These constraints ensured that any expression created by the structure editor, whether complete or not, was syntactically valid. The structure editors also made it clear from any given expression what the valid substitutions were and allowed users to select rather than type often complex function and object.slot names. Lastly, the structure editors formatted the rules and functions in order to give them a uniform appearance. For the most part, these features were positively received and helped the language and environment meet many of the users' goals. There were, however, some drawbacks to these features, which will be discussed in the following sections.

6.3.1.1. Editing Constraints

The manner in which the structure editors constrained how users created and edited expressions was discussed in Section 5.7.3.1. While these constraints ensured that the rules and functions were always syntactically valid, they could also make editing difficult for those who were used to free-form text editors. A good example of this came to light during one of the user tests. In this example, one of the users wanted to create a complex logical expression. The expression was to be used as the antecedent of an `IF` expression and is shown in Figure 60. The original and much longer names have been substituted with simpler names due to space considerations.

```
MBD > TTD OR TTD - BLDC <= SJD[] AND TTD - BLDC <= MLOD
```

Figure 60. Simplified logical expression.

The creation of this expression required the use of the palette to change the initial unspecified expression one step at a time until the final expression was reached. An example of one such substitution sequence is found in Figure 61. The underlined expressions are those that get replaced in the next expression and, for simplicity, this sequence fills in the values for terminal expressions as soon as possible. While the use of the palette was the intended method of creating syntactically valid expressions, the users were used to free-form text editors and initially tried to create the expression

starting from the left-most expression, `MBD`, and work their way to the right. Using the structure editor, this was prohibited because the left-most expression was a numeric expression and the antecedent of an `IF` expression had to always be boolean in order to be syntactically correct. While the user could, in theory, open an in-line editor to create this entire expression directly, the size of the in-line editing field, the length of the names of each of the subexpressions and the complexity of the entire expression would have made this difficult.

```
<expr>
<expr> OR <expr>
<expr> OR <expr> AND <expr>
<expr> > <expr> OR <expr> AND <expr>
MBD > <expr> OR <expr> AND <expr>
MBD > TTD OR <expr> AND <expr>
MBD > TTD OR <expr> <= <expr> AND <expr>
MBD > TTD OR <expr> <= SJD[] AND <expr>
MBD > TTD OR <expr> - <expr> <= SJD[] AND <expr>
MBD > TTD OR TTD - <expr> <= SJD[] AND <expr>
MBD > TTD OR TTD - BLDC <= SJD[] AND <expr>
MBD > TTD OR TTD - BLDC <= SJD[] AND <expr> <= <expr>
MBD > TTD OR TTD - BLDC <= SJD[] AND <expr> <= MLOD
MBD > TTD OR TTD - BLDC <= SJD[] AND <expr> - <expr> <= MLOD
MBD > TTD OR TTD - BLDC <= SJD[] AND TTD - <expr> <= MLOD
MBD > TTD OR TTD - BLDC <= SJD[] AND TTD - BLDC <= MLOD
```

Figure 61. Possible logical expression creation sequence.

Another related issue was how the creation order of expressions affected their meaning. For example, the meaning of the expression in Figure 60 would change depending on which logical operator was entered first. If the user had created the `AND` part of the boolean expression and then added the `OR` part, the meaning of the expression would have changed to what is in Figure 62. While the parentheses would have been placed in the expression to help make it clear the precedence order, there was no easy way to change this precedence order if the user felt a mistake had been made. The only way, in the context of the structure editor, was to remove the entire expression and recreate it starting with the `OR` part of the expression first.

```
MBD > TTD OR ( TTD - BLDC <= SJD[] AND TTD - BLDC <= MLOD )
```

Figure 62. Modified logical expression

While these constraints could make the use of the structure editors difficult at times, it is unclear if these problems were serious enough to warrant finding alternative approaches to the creation of rules and functions. The structure editors worked well and allowed the users to create whatever expressions they needed. In addition, the users could edit the ruleset file directly using a standard text editor if they wanted to. While this was not encouraged, it was not prohibited. One possible solution, however, would have been to remove the syntactic restrictions from the structure editor. This would have allowed expressions to be created using the language's available operators and any operands associated with the currently loaded *RiverWare* model. In order to insure that syntactically invalid rules and functions were not permitted, the users would have to verify the correctness of these elements before they could be used. This would provide functionality that was a cross between that provided by the structure editor and that provided by the in-line editor. While it would not have been difficult to have allowed this functionality and it would have provided some potentially useful flexibility, it is not clear that the cost of removing the syntactic restrictions, especially given the inexperienced user population, would have been beneficial. Based on many of the users' experiences with the previous language is it easy to imagine them creating syntactically invalid rules and functions and finding it difficult to understand why they were not valid and how to fix them.

6.3.1.2. Selecting Expressions

As the design of the structure editor progressed, I needed to come up with a simple means of selecting expressions. I decided to use a single mouse click since this was the most straightforward approach. Thus, for a terminal expression, the user had only to place the cursor over the expression and click the mouse. This would select and highlight the expression. Selecting complex expressions, however, presented a problem, since I needed to allow the user to select the entire complex expression as well as any sub-expressions that might be part of it. Given that I did not have a lot of time for programmatically difficult alternatives, I decided to allow the user to select the complex expression by selecting its operator, in the case of unary or binary expressions, and any of its keywords/symbols, in the case of control structures such as the `IF` expression. While this worked, was consistent and

unambiguous, it was not immediately intuitive. In addition, it was sometimes difficult for users click on small operators, such as the minus sign. On the whole, however, this was the best approach given the time constraints I was under. Another approach that I considered, but did not have the time to implement, was using click and drag to highlight the selected expressions as they were captured. I suspect that this would have been useful, although not necessarily trivial to implement.

6.3.1.3. In-line Editing

As discussed in Section 5.7.2.2.2, in-line editing was provided as a means of allowing users to add and update expressions without having to use the palette. This was initially intended as a way to create and edit terminal expressions, but was also used on other, more complex, expressions. While it worked reasonably well, there were a number of problems that made it less useful than it might have otherwise been. One problem related to the manner in which terminal expressions were generally edited and required the user to perform up to five mouse clicks. An example scenario was when users wanted to select and alter a string that contained spaces. They would need to double click the expression to open the in-line editor and then perform either a triple click or a click and drag to select the entire expression. This problem was originally identified during the pre-user testing when a user complained that his hand was getting tired from all the clicking. The problem was rectified prior to the full user test by having the entire expression selected within the in-line editor by default. This allowed the user to open the in-line editor and have the entire expression selected with just a double click.

Another problem with in-line editing was related to how invalid expressions entered by users were handled. Each expression entered during in-line editing was parsed prior to its acceptance into a rule or function to ensure that it was syntactically correct. If the entered expression was invalid, an error message would be displayed, the in-line editor would be closed and the original expression would be returned. As a result, the user would be required to reenter the new expression. This could be particularly annoying if the new expression was complex and the mistake trivial. Instead of reverting to the original expression, the in-line editor could have remained open with the invalid expression in it so that the user could make the changes needed to make it syntactically correct. Unfortunately, this option

would also have a drawback associated with it. It would not permit the user to return to the original expression without retyping it.

Related to the above topic is how an in-line editing session had to be cancelled. Acceptance of an in-line editing session required a carriage return, which was reasonably intuitive. Rejection, on the other hand, required the user to single click the mouse somewhere outside of the in-line editor's text field. While this approach could be documented and would probably be stumbled upon by users, it was probably not the best approach. A better approach, and one that would also solve the problems discussed in the previous paragraph, would have been to replace the in-line editing text field with a simple modal dialog box that would contain a text field for the expression to be edited and options to accept and cancel the changes. It could be invoked just as the in-line editor was and could allow for resizing of the text field.

6.3.1.4. Use of Colors and Fonts

Initially, I had planned on using colors and fonts in expressions to provide visual cues regarding the various parts of an expression. For example, keywords would be displayed in black, Arial font, while terminal expressions would be displayed using Times Roman font and color coded based on their type: blue for numeric, maroon for boolean, green for string and so on. These visual cues could in turn be echoed on the palette where each of its expressions could be displayed to reflect their return type. For example, relational operators would be displayed in Times Roman and colored maroon. I partially implemented this by making all keywords and operators black and all terminal expressions blue, but did not have time to implement the rest. While I suspect that this would have been useful, I am not entirely sure. It is possible that given all the types that the language supported the editors would have looked too busy. On the other hand, it might have made the identification of valid substitutions more obvious to the novice user. Had time permitted, it would have been interesting to try this out and see how it affected the use of the structure editors.

6.3.1.5. Function Parameter List Editing

The rule and function editors made consistent use of structure editing for creating and updating expressions. The main exception to this, other than in-line editing, was the editing of a function's parameter list. This was accomplished in a manner that was similar to that used for in-line editing. In order to edit a function's parameter list, the user had to type directly into a provided text field and, when finished, enter a carriage return. The parameter list would then be parsed for syntactic correctness and accepted if correct. While this approach worked, it was not consistent with the structure editing supported elsewhere. It required the user to know the syntax of the parameter list, which although not complex, would not necessarily be intuitive to a non-programmer. Even though I realized this at the time, I did not have time to add the needed structured editing capabilities.

6.3.1.6. Line Wrapping

The structure editors maintained a consistent look and feel for the rules and functions that they displayed. For the most part, this was a good thing. The main exception involved the representation of very long expressions. The structure editor did not break these expressions into multiple lines automatically and did not provide a way for users to break them into multiple lines. As a result, long expressions would be displayed off the right hand side of the editor window. To mitigate this somewhat, the editor window included a horizontal scroll bar so that the user could view the entire expression. There were a few problems with this, however. One was that it was sometimes not obvious when the expression continued off the right hand side of the editor window. Another problem was that even if the window were enlarged to its maximum size, there could always be expressions that would be longer and using a scroll bar to view parts of an expression would not always be as useful as being able to view the entire expression at one time.

This was a known problem prior to user testing and one that also surfaced during user testing. There were a few solutions that could have been used had time permitted. One was to set a maximum length for a line and automatically wrap the line. While this option had the advantage that it would help to maintain a consistent appearance for all expressions, it would be difficult to design a generic solution

that would never split an expression in a nonintuitive manner. Another alternative would be to allow users to specify where an expression should be split. This would have the advantage that the users could determine what made the most sense, yet could lead to the same rule or function having a different appearance based on user preferences. In addition, the point at which the expression needed to wrap would somehow have to be included in the language since it would have to be saved with the file and restored on subsequent editing. Given this, the first solution would be preferable, although it would most likely be difficult to implement.

6.3.2. Functional Consistency for Editors

Each editor was responsible for allowing users to view and edit one major language element. Yet, there was also functionality that was common to all editors or at least could be used by more than one editor and still make sense. For instance, each editor allowed for the user to save the ruleset file without having to use the main ruleset editor. While this worked as intended, there was some confusion as to what was being saved. In one instance, a user thought that the save option on the function editor would save just the contents of the function. This was corrected by changing the wording to make it clear that the save option pertained to the entire ruleset.

Another area in which common functionality was used was for syntax checking. Each editor allowed the user to verify that the syntax of the current language element was correct and complete. Thus, when invoked from the function editor, the function would be checked and when invoked from the group editor, the group's rules and functions would be checked. This would tell the user whether or not the language element and its contents could be used by *RiverWare* during a simulation.

6.3.3. Mouse Click Consistency

Single and double clicks were used throughout the programming environment to various ends. They were used to select items, display the contents of groups, open editors and toggle the state of fields. While these were all useful functions of mouse clicks, the rules for using mouse clicks were not always consistent. This became apparent during user testing when one of the users expressed confusion

regarding when single clicks and doubles were required. For example, when using a ruleset or group editor, a single click on the name of a group, rule or function opened an in-line editor that could be used to change the item's name. Yet, when using a rule or function editor, a double click on an expression was required to open an in-line editor. A single click would simply select the expression. The editors were different types, but given that both opened in-line editors, the use of a different number of clicks proved confusing.

In order to avoid confusion, a consistent meaning for single and double clicks should have been devised. For instance, single clicks could be used to select things, such as groups, rules and functions from ruleset and group editors and expressions from rule and function editors. Double clicks, on the other hand, could invoke some kind of editing. This would work in most cases without any operational conflicts. An exception would be for the ruleset and group editors in which a double click on a line could open a corresponding group, rule or function editor and open an in-line editor to change the name of the corresponding group, rule or function. In order to maintain a consistent standard, it would probably be necessary to remove the in-line editing functionality and provide an alternative means of changing a group, rule or function name.

6.3.4. Language and Presentation Accessibility

The use of terms that are familiar to target users is not particularly profound, but is nonetheless important. There were two main areas in which this was a problem in the second language and its environment. The first was in the use of computer science terms on some of the environment's screens. An example of this is found on the palette. In order to provide some kind of logical grouping of the different operations and control structures that were part of the language, the palette grouped them and placed the groups in titled boxes. For example, all unary expressions were placed in one box with the title "Unary Expressions" and all binary expressions were placed in box with the title "Binary Expressions." While this kind of grouping and their titles made sense from a programmer's perspective and even reflected the manner in which the operators were internally grouped, they tended to be

confusing to users who were not as comfortable with computer science terms and with the grouping of operators based on their number of arguments.

A better solution would have been to group the operators based on their function and title them accordingly. For instance, all numeric operators (i.e., those that took numbers as their arguments and returned numbers as their results) could be grouped under “Numeric Operators.” Likewise, all operators that acted on and produced DateTime types could be grouped and all relational and logical operators could be grouped and titled “TRUE/FALSE Operators” or “Conditional Operators.” This solution would not be perfect since some operators were overloaded. “+” and “-,” for example, were shared by Numeric and DateTime types. To address this problem, one of two solutions would be possible: provide different operators to be used with DateTimes or place copies of these operators in the DateTime box as well as the Numeric box.

6.3.5. Multiple User Collaboration

During the first user test, users decided to split into two groups, with one group creating the rules and another creating the functions that would be used by the rules. Although I had not anticipated the users wanting to use the programming environment in this manner, this type of multiple user collaboration was supported in an indirect manner. To do this, the users who were creating the rules would have to also create placeholder functions for any functions that the rules directly used. The users who were creating the functions would also create these same, albeit fully specified, functions in addition to any others that might be needed. Then, when it came time to combine the rules and functions, the users could open both rulesets, discard the placeholder functions and combine the rules and functions. This approach worked reasonably well and was what the users did.

While this may not have been the most elegant solution available, it was reasonably simple and allowed for largely unrestricted collaboration. An alternative solution would have been to provide the ability to edit a shared ruleset. Given that this would have been rather time-consuming and would not add a significant amount of functionality, it should only be added if there is a compelling need for it.

6.3.6. Ruleset and Group Editors

When I began the design of the programming environment, I had assumed that each of the major language elements – rulesets, groups, rules and functions – would have their own editor. I even explored the idea of having individual editors for the rule statements, although I rejected this idea. While one editor for each of the major language elements made sense and worked well for the rule and function editors, the manner in which the ruleset and group editors were developed was less than perfect. The problem lay primarily in how the two editors' functionality overlapped. The ruleset editor allowed for almost as much editing capability as was available using the group editor. I had not originally intended on doing this, but little by little started including the group editing functionality in the ruleset editor. In hindsight, it would have been better to restrict the editing options for the ruleset editor to the contents of the ruleset. The group editor would then be the sole means of editing the contents of groups. This would have decreased the development time without losing any functionality.

6.3.7. Important Functionality First

Early in the design of the programming environment, I became aware of Hudson's work that described the benefits of having screens that displayed the important information at all times and allowed the user to enlarge the screen to show ancillary information when desired [Hudson, 1994]. I decided to use this approach for the language's editors, because they would need to be able to display a fair amount of information. Some of this information was not critical and there were likely to be many editors open at any given time.

After quite some time and effort, I completed the functionality, which worked as intended. Yet, the amount of time it took was significant. The user interface libraries that were used for *RiverWare* and the rule language's programming environment did not directly support this functionality and actually made it difficult. As a result, time was spent adding functionality that, while useful, was not critical. This time could have easily been spent on more important functionality, including performance improvements or more intuitive error handling and debugging.

The lesson here is that additional functionality should be prioritized with respect to the amount of effort required to add it. Had this functionality been easy to add, it would have been worthwhile to do so. As it was, the large amount of time spent was not justifiable given the non-critical nature of the functionality. In hindsight, a small amount of time should have been spent in order to get an estimate of the effort it would take to add the functionality and, when it had become clear that the solution was nontrivial, alternative approaches should have been considered. Examples would have been to display all information in larger editors or display just the critical information and use modal dialogs to display the ancillary information.

6.4. Design Methodology

This section will discuss some of the issues related to the design of an application-specific programming language and environment.

6.4.1. Language Design is Difficult

The design and development of the language and its accompanying environment turned out to be considerably more difficult and time consuming than I had originally anticipated. This was partly due to the fact that I was not an experienced language designer and partly because I was rather ambitious in my design. In all likelihood, my lack of design experience contributed to my ambition. While I do not regret the resulting language and programming environment, I would caution future language designers and developers to carefully consider time and budget constraints. These time and budget constraints need to be taken into account given the abilities of the designer and developer. For example, I am confident that more experienced language designers and developers could have created a superior language in less time than it took to create the *RiverWare* rule language.

What then, should be done when it appears that an application-specific language is needed? The first thing, as was stated previously, is to assess whether or not a language is truly necessary. Depending on the problem, it is possible that a programming language may be excessive and that a set of predefined functions or a simple interface might suffice. Even if a language is deemed necessary, it is

possible that an existing language can be borrowed. For example, Tcl, Python and Perl can all be embedded within an existing application as was done for the first language. In addition to being embedded within an application, these languages also allow for a high degree of extensibility so that most features that are needed can be added. Of course, all of these languages are either procedural or object-oriented, which may not be the best paradigm from which to work. However, as was also discussed in the previous chapter, an interface could be overlaid on top of the language if desired and would more than likely be less work than creating a language from scratch. In short, unless there is a compelling reason to design and develop your own language, it is probably wise to consider and reject other options first.

6.4.2. Walkthroughs

Programming walkthroughs with groups of target users proved valuable during the design and development of the language. One example in which a deficiency in the language was identified occurred during a presentation to a group of eventual users. Part of the presentation was a walkthrough of some of their rules that included both what the language might look like and how it might be edited. During the presentation, a discussion on the means of prioritizing rules ensued. Given that the previous language had prioritized all rules in a ruleset with respect to each other and the users of that language had never requested anything else, I designed the prioritization of the rules in the second language similarly. During the walkthrough, however, it became clear that this was not sufficient. While one set of users wanted the rules to be prioritized as before, another group felt the rules should be prioritized within their groups and then the groups should be prioritized with respect to each other. This latter option would allow the users to swap the priorities of groups and thereby reprioritize a whole set of rules. After it became clear that both sides needed the functionality that they had requested, it was decided that both means of prioritizing rules should be included in the language. The walkthrough helped to identify the problem early enough so that it could be corrected during the design phase rather than after development had started.

6.4.3. Problem Descriptions

The problem descriptions used in the second case study were the Tcl-based rules that resulted from the first case study. They were really just complete versions of the original policies that the users had given me at the start of the first case study. As a result, the two sets of problem descriptions were conceptually identical. This section will discuss some of the factors related to the use of these problem descriptions.

6.4.3.1. Relationship to Goals

The design goals in combination with the problem descriptions helped to drive the design of the second language. This helps to explain why even though the problem descriptions used in the second language design were represented procedurally, the resulting language was primarily functional. This is in contrast to the first language in which the problem descriptions were represented informally and non-procedurally and the resulting language was procedural. The objective of meeting these goals helped to drive the design of the second language in a different direction than the first language.

6.4.3.2. Multiple Representations

The users' problem descriptions had a number of different representations at the time I started the design of the second language. They included the original, FORTRAN-based *CRSS* code, the Tcl-based policies written in the first language and the original, paper-based descriptions that the users had provided. What is interesting here is that while the first two were more completely specified and more closely represented the *CRSS* policies, the third representation was probably the most important given the users' goals for the language. For while the first two were logically more correct, the third was representationally more correct. It described the policies in a way the users thought about them, largely unconstrained by the syntax of a particular language and contained a mixture of equations, tables and logic-based English.

Highlighting the third representation does not imply that the logic and even the representation of the first two were not of interest. They were insofar as they provided a look into the kind of problems

that the users would need to be able to solve using the language. Unfortunately, these representations did not make it clear to someone unfamiliar with the languages what the policies really meant. As a result, while they provided a representation of the policies that was good enough for the computer to use, they did not provide a representation that made it easy for a human to read.

6.4.3.3. Incomplete Problem Descriptions

The problem descriptions provided by the users are likely to be incomplete because the users are unlikely to be able to anticipate everything that they will need to represent using the language. This was certainly true for the first language's design effort and was, to a lesser degree, true for the second. While it would be ideal to be given a complete and correct set of problem descriptions, this is unlikely. And even if it were to happen, it is probable that the users would develop new ways to use the language and new users would want to use it in ways that were not anticipated by the designers or the original users.

Given this, the designer must take these descriptions and make sure that any requested functionality is generalized as much as reasonable. This is not to say that the resulting language should be a general purpose language, although this is certainly one possibility. It is more that the problem descriptions should be taken as the users' best guess as to the functionality needed and not a complete description of the desired language. As a simple example, if arithmetic is to be part of the language it would not make sense to exclude commonly used operators because they do not happen to be included in the problem descriptions.

In the new language design, I was trying to avoid a general purpose language given the problems that the users had had with the previous language. Yet, given that I knew the language would be used in ways that were beyond the problem descriptions, I was worried that I would omit functionality that the users would need. This led to the addition of external functions in which the users could create and call functions written in the previous, Tcl-based language. This allowed the users to represent anything in the new language that they had been able to using the previous language and

provided me with the latitude to proceed with the design knowing that any functionality omitted from the new language would not cause the users to be unable to represent their policies.

6.4.4. Design Goals

The design goals were an important part of the design of the new language. For unlike the design of the first language, I had a better sense of what the users needed to do with the language and better understood some of the important aspects of a programming language intended for non-programmers. Instead of focusing primarily on expressiveness and writability, I also made sure that the language was readable, maintainable and safe. These goals helped to place the problem descriptions in a context that made it clear what the users needed. Rather than looking strictly at what they had given me, I looked also at who would be using the language and how.

It is difficult to exactly state how the design goals contributed to the design of the language. Rather than driving major decisions, they set the tone for the design and were used to make decisions when different alternatives were being considered. For example, when it was clear that the language would use an internally developed editor, the choice had to be made regarding the representation of arithmetic expressions. One option was to display the expressions linearly, as they would be in a text editor. Another was to display them as the users thought about them and wrote them, for example, representing division with the numerator over the denominator and separated by a line. While the former option was far simpler to implement, the latter option supported readability better and was thus chosen.

6.4.5. User Documentation

Little to no documentation was provided as part of the language delivery or user test. This was largely a matter of time as I barely had time to complete the language and environment functionality necessary to allow for meaningful user tests. The lack of documentation was certainly a drawback and this was notable on a number of occasions during the user test. While extensive documentation may not

have been needed, it would have been helpful to have documentation to explain and show examples of the functional nature of the language and the way the structure editor worked.

The main motivation for having good documentation was to familiarize the users with those parts of the language and environment that were new to them. For the language, this would include the functional nature of the language, the main language elements, the use and limitations of external functions and the set of predefined functions available. For the programming environment, this would include an overview of much of the provided functionality given that the entire interface was new. The main focus would be on the creation of each major language element, the functionality provided by each editor and the use of expression selection and the palette. Ideally, this would be sufficient to allow the users to gain enough familiarity to create and edit rules. If not, I suspect that this would indicate that the language and/or the environment was too complex for the targeted user population.

6.4.6. User Test

The user tests were helpful in gaining an understanding for how a representative sample of users perceived and made use of the language and environment. They provided insights into those parts of the system that they found intuitive and those that they struggled with. These insights were somewhat exaggerated by the fact that the users were provided with only a brief training session and very little documentation. While it would be interesting to investigate how additional training and documentation would affect the users' experience with the system, these early user tests were helpful in gaining an understanding of how they perceived a system that they should ideally be able to use with little guidance.

The user tests were not without their faults, however. One significant problem, especially in the first user test, was the number of bugs that were found by the users. While it was not surprising to find these bugs given that this was the first set of users who actually used the system, the users got understandably frustrated at times. In addition, the presence of bugs caused the user tests to be more disjointed than they would otherwise have been since encountering a bug would require anywhere from

a work around to restarting *RiverWare*. In one particularly unpleasant bug, one of the users lost his entire ruleset.

In order to minimize the number of bugs found, it would have been wise to have had a "friendly" user test that would be conducted before any other user tests in order to eliminate as many bugs as possible. While I had received feedback on the system prior to the first user test from a single user who had spent some time using it, this information was reported to me rather than gained observationally and contained mostly minor feature requests. It did not address any of the serious bugs that the first set of users encountered and, even if it had, it would have been difficult to fix them without knowing the context within which they occurred.

In addition to an initial "friendly" user test, it would be valuable to perform user tests that addressed specific issues related to the language and environment. These could provide insight into user comprehension of rules and functions written by others in the new language, the ease and frequency of use of external functions, the ease of use of all editing functionality not supported by the structure editor (e.g., function parameter list editing), the ease and frequency of use of the `FOR` and `WHILE` loops and the ease of use of unit checking and conversion. Some of these, for instance the ease and frequency of use of external functions, might not be amenable to a short term user test. They might be better served by a regular audit of the users' rules and functions in order to determine if the users were continuing to use external functions and, if so, if they were creating new ones or simply failing to transcribe the old ones into the new language.

Chapter 7

User Feedback on the Language and Environment

The second language and environment have been in use within *RiverWare* for approximately four years. They have been used by USBR users who provided the original set of problem descriptions and who provided design and use feedback for both languages [e.g., Fulp and Harkins, 2001]. They have also been used by those from other USBR offices and users from the Tennessee Valley Authority. Given the amount of time the language and environment have been used and the variety of users who have experience with them, it is appropriate to solicit feedback about the users' experiences and to get a sense for their strengths and weaknesses. To this end, I interviewed Terry Fulp, who has been the sponsor of the language and environment since its inception. He has worked with CADSWES and *RiverWare* or its predecessors for more than a decade. This section will convey and discuss the feedback that I received from Terry regarding the language and environment. It will give his high level views followed by specifics related to the language and environment.

7.1. Overview

At a high level, the language and environment are working well. The original *CRSS* ruleset has been re-coded in the new language and the results it generates are now considered authoritative. In fact, the USBR now refers to this ruleset and its accompanying *RiverWare* model as *CRSS*. In addition to those modeling the Colorado River, there are other groups using the language. An example is a group of users from the USBR office in Albuquerque that is using *RiverWare* and the rule language to model the Rio Grande. These users were not part of the original user group and their policies were not part of the original problem descriptions, yet they have been able to make effective use of the system. In addition to these users, the language and environment are also used by non-programmers to maintain and create rules. One example is a new employee from one of the USBR's regional offices who is a water resources

engineer with no programming experience. He attended the *RiverWare* rulebased simulation training course, an early version of which was used as the context for the second set of user tests, and has been able to use the language and environment with little help or supervision. This has included updating and creating rules and functions and running simple simulations.

7.2. Language

7.2.1. Functional

The functional nature of the rule language is considered a positive characteristic overall. One of the primary advantages of the language is the ability to encapsulate details of a policy within its functions. Thus, a rule can express the policy at a high-level, which may be sufficient for many users, while still allowing the details of the functions to be explored to whatever depth is necessary. One of the main drawbacks of the language, however, is that it generally takes new users a while to learn how to make effective use of it. Generally, these users have little programming experience and the experience they do have is typically procedural. As a result, it often takes them a while to translate their problems into the form required by the language. Once they get used to the language, which is typically fairly rapid, they are able to become proficient using it and grow to like it.

7.2.2. Readability

The rules and functions written in the new language are generally more readable than their Tcl-based counterparts. This is partly due to the functional nature of the language and its associated ability to hide the details of policies within a rule's functions. It is also partly due to a cleaner syntax. The Tcl-based rules were difficult to read and understand because of the large amount of extraneous detail that preceded the core of the policy and because of the language's awkward syntax. This is not to say that all the rules are equally readable, since this depends somewhat on the rule writer. If the rules are created in a monolithic fashion and do not rely on the consistent use of functions to encapsulate ideas, they can be more difficult to read than if they are well structured. This is not unexpected, since any language can be made difficult to read by employing poor programming practices.

The rules are read by programmers and non-programmers. Both groups are water resources engineers who understand the policies at at least a high level. The programmers generally spend time creating and modifying rulesets and using them in the context of *RiverWare* rulebased simulations. Because there are programmers who are responsible for the rules and related functions that apply to their portion of the river basin, users often share portions of their rulesets with each other in order to run full basin simulations using a complete and up-to-date ruleset. The non-programmers are usually water resources engineering managers. They have generally neither written nor modified any rules, but can understand them. This is in contrast to the rules written in the Tcl-based language, which they might have been able to understand with some difficulty, but would not have spent the time to do so.

Rules are also sometimes shown to those who do not use *RiverWare* at all. These people are often policy makers and the discussion of the rules generally stays at a very high level. On occasion there is a need to dig deeper into the rules and in these cases there is often a need to explain the meaning of the details that are found within the functions.

7.2.3. Sharing

The sharing of rules and functions between rulesets is not as easy as the users would like it to be. This is largely due to the fact that the new language stores all rules and functions associated with a ruleset in a single file. There is no way to reference rules or functions that exist outside of a ruleset file. This has resulted in the users sending portions of their rulesets to each other via email. While the programming environment makes the integration of the mailed rules and functions easy, it is still a manual task and there is always a chance that the users will not be using the most current version of the rules and functions.

The previous language made the sharing of rules and functions easier because it required rules and functions to be stored in their own files. As a result, users could create a ruleset that referenced a common set of rules or functions. It also led to a proliferation of files that were difficult to maintain. In addition, this ability to share was limited to rules and functions stored in files on locally available file

systems. Given that the bulk of the rule and function sharing that is required is among those who are located in geographically dispersed offices that do not share a common file system, the approach taken by the previous language would not work for many of the users.

The obvious solution to this problem is to allow portions of a ruleset to be stored in their own files, as was allowed in the first language. To avoid the problems associated with a ruleset consisting of too many files, the referencing of external files could be made optional. Thus, a ruleset could contain all of its groups, rules and functions internally or it could reference any portion that it needed to. To allow for the sharing of these ruleset portions across separate file systems, references to publicly available servers, such as web servers, could be supported. This would allow a single ruleset to exist across multiple file systems and allow groups to maintain the portion of the ruleset for which they were responsible. A solution such as this, of course, requires cooperation to insure that the publicly available ruleset portions are always available, up-to-date and bug free.

7.2.4. Dependencies

The automatic determination of dependencies has worked as intended. It has provided a means of making sure that the rules are executed only when needed. Recall that the former system required the users to specify the rules' dependencies. This had two drawbacks. The first was that it was difficult for the users to know which object.slots should be made dependencies and any mistakes could cause rules to be executed more or less often than they should be. The second was that some of the users purposefully manipulated the dependencies, and thus the number of times the rules executed, to get results that they felt were correct. Both of these could lead to incorrect results and could make the policies the rules were representing unclear.

The automatic determination of dependencies addressed the first problem and, I thought, the second. As it turned out, however, the users have found other means of manipulating the practical order and frequency with which rules are executed. They use the rules' execution constraints and conditional assignment statements to determine which values will be set and when they can be set. This is an

improvement over the misuse of dependencies, because the logic is explicitly stated in the rules themselves and the results will be entirely consistent with this logic. In addition, given the form of the language, this type of manipulation is sometimes necessary. For example, there are times when users must make sure a rule is executed only once per timestep. To accomplish this, they will create a rule execution constraint that checks that the value the rule to be set has not already been set. An example of this is shown in Figure 63. In this example, if `Starvation.Storage` has not been set for the current timestep, the rule will execute and try to set it. If, on the other hand, `Starvation.Storage` already has a value for the current timestep, the rule will not be executed.

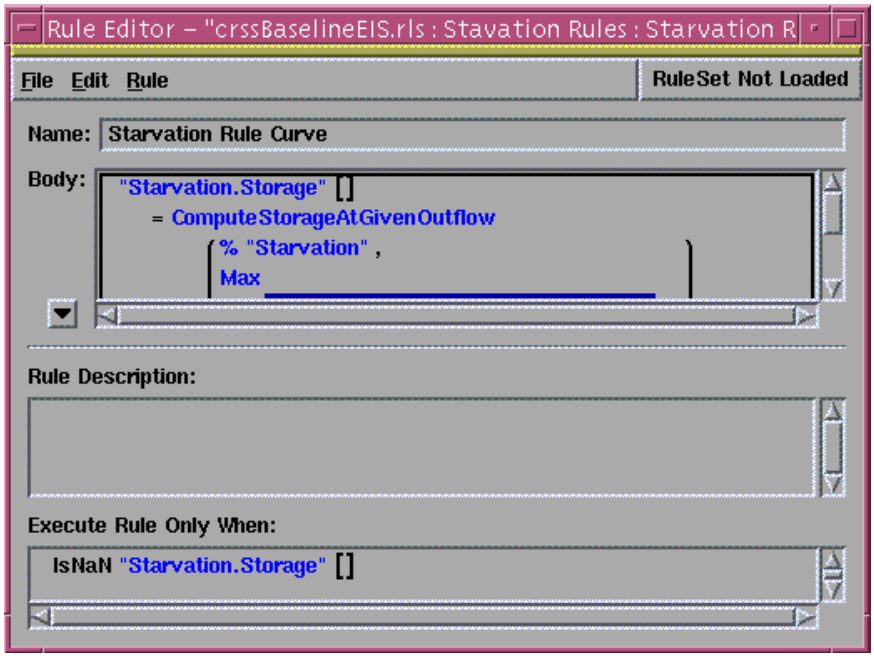

Figure 63. Single execution rule.

There are times when this mechanism is overused, which can result in complex rule execution constraints and, by extension, difficult to understand rules. This has mostly occurred in the rules that have been translated from the Tcl-based language to the new language. Since the Tcl-based rules had relied on the manipulation of dependencies and the users wanted to generate the same results, they did so by making the new rules act just like the old rules. While this has worked to a degree, users have created more complex scenarios and found that the original Tcl rules were not as complete as they

thought they were. As a result, in some cases, they have determined that they need to rewrite the rules and not make such heavy use of the rule execution constraints.

Another reason users have resorted to the use of rule execution constraints and conditional assignment statements is that they are afraid that rules or assignment statements that are meant to execute once per timestep will be executed more often because of the automatic determination of dependencies. They generally don't check to see if this is actually the case, because it is easier to set the constraint or create the conditional assignment.

7.2.5. Rule Structure

Object.slot values can only be set from within the body of a rule. This makes it clear which values a rule can set and under what conditions they will be set, which can in turn make clear the intent of the policy that the rule is representing. This mechanism for setting object.slot values during a rule's execution, however, is the only means of sharing information between rules or between executions of the same rule. As a result, users often find that they need to use this mechanism to set values that are not strictly part of the main body of the policy. This can result in rules that are cluttered by excessive assignment statements, which can obscure the intent of the rule. An example of where this is necessary is the calculation of forecasted inflow, which is used in a number of rules. This value must be calculated once at the beginning of each timestep and not recalculated again until the next timestep. At present, the only way to accomplish this within the rule language is to set this value using a conditional assignment statement where the condition is only true if the value has not been set.

While this allows users to create rules that produce the correct results, it does so at the expense of the rule's readability. As a result, when they need to explain the rule to others, they find that they have to explain away the presence of these extra assignment statements. Therefore, an alternative solution should be found. While it may be possible to calculate these values outside of the rule language using other *RiverWare* facilities, these values are really part of the policy. Accordingly, they should remain as

part of the ruleset, as this will likely aid future maintainability by keeping all calculations associated with the rules in the rule language.

Two of the more straightforward solutions involve simple extensions to current ruleset elements. The first is to allow the rule writer to express these shared values as functions whose frequency of execution can be constrained. The default could be that functions are re-executed each time they are invoked. Functions could also be constrained to only re-execute once per timestep or once per simulation. In these cases, subsequent calls to the function within the specified time period would yield the result calculated from the function's initial execution during that time period. While this would work, it would not provide a means of saving the values calculated to an object.slot. For more information on this topic, see Section 7.3.6. In addition, this solution might make the rules more difficult to understand, since the behavior of functions would depend on how they are configured.

Another option is to provide a utility rule that can hold these ancillary assignment statements. This rule would behave like a normal rule in most respects. It would be called only if its dependencies were updated and would make use of rule execution constraints. There would, however, be two significant differences. One would be that the setting of any of its values would not be sufficient to cause control to return to the simulation engine. The other would be that this rule would have a higher priority than all other rules so that in cases where it and other rules needed to be executed, it would be executed first so that the subsequently called rules could make use of any values that it calculated. It is unclear if either of these two solutions would solve users' problems without introducing others or opening up the language to misuse. To this end, it would be wise to perform some code walkthroughs with users to get a sense for how either of these two options would be used in practice.

7.2.6. Control Structures

The control structures that were included in the language have not been particularly well received. While there are some who have become proficient with them, most find them difficult to learn and cumbersome to use. Surprisingly, for the *CRSS* ruleset, this has not led to the use of external

functions in places where these control structures are needed. Instead the core language has been used, although most people tend to avoid the portions of the ruleset that make use of these control structures. For the *CRSS* ruleset, one programmer in particular has made it a point to use the core language wherever possible and has conducted the bulk of this programming. While the users were opposed to the use of recursion during the design of the second language, they now feel open to considering it.

7.2.7. External Functions

External functions are rarely used in the existing rulesets. They were used extensively to allow for a smooth conversion of the *CRSS* rules from the Tcl-based rule language to the new rule language. With only a few exceptions, however, all of these external functions have been converted to corresponding rules and functions. These exceptions have been left as external functions because the users have not gotten around to converting them rather than their inability to express the code in these external functions using the main language. The fact that external functions are infrequently used came as a surprise; I had expected to see some of the more complex logic in the rules remain as external functions and at least some of the complex new functions coded using external functions. There seem to be two main reasons that this did not happen. The first is that the original users have made a clean break from the original language, at least in part due to the one user mentioned above who made it his goal to completely convert the old rules to the new language. The second is that the old language is not taught as part of the training course and new users are not encouraged to use it. As a result, there is little chance that new users will ever become familiar enough with the old language to make use of it.

7.2.8. Unit Conversion and Checking

Automatic unit checking and conversion are considered two of the best features of the language. They make the language both safer and more convenient with regard to the values used by the rules and functions. The language is safer because rules and functions are unable to compare or combine values with incompatible units. The language is more convenient because, with the exception of literals, users do not have to concern themselves with a number's scale and units, which are automatically transferred to the value's source. The value's source may be an object.slot, a function or an arithmetic

expression. This functionality is in contrast to the previous language and most general purpose languages that allow for unrestricted combining and comparison of values without regard to their scale and units.

7.3. Environment

7.3.1. Multiple Editors/Rulesets

The use of multiple editors and the ability to have multiple rulesets open for editing has worked well. Users have been able to keep track of the various language elements that they are editing and have found that the color bars are helpful in matching editors to rulesets. They have also made use of the ability to copy and paste elements between rulesets. This ability has allowed them to share portions of their rulesets with each other. As was discussed in Section 7.2.3, this is partly due to the fact that the rule language does not support explicit sharing of rule elements.

7.3.2. Structure Editors

Users are happy with the structure editors and palette. They find that the top-down editing style works well and that it only takes a little time to get used to this approach. They prefer the structure imposed by the environment to free-form editing, since it helps them avoid mistakes. Terry, in particular, works by performing frequent validity checks while using the structure editors to create functions and rules in order to insure that they are complete and correct. He also uses existing rules and functions as templates by making and modifying copies of those that are similar to rules and functions he wishes to create. This is akin to the approach that I had envisioned would be used with the clipboard, which I had intended on making part of the environment. The main distinction is that the editing is performed in the new rule or function rather than in an external workspace. This is a better approach given that it can be done in the context of the new rule or function and it required no additional development time.

One interesting point regarding the use of the structure editors and palette that came to light while I was using *RiverWare* to get screen shots is that the palette does not always enable the correct set

of substitutions for a given selection. This is a bug that appears to have been introduced some time ago and results in the users being able to create syntactically invalid rules and functions. It is surprising that no one at CADSWES was aware of this bug given that its existence severely compromises the usefulness of the structure editor. While I can't be sure, I suspect that the reason that this bug was not noticed is that it does not prohibit the creation of syntactically valid rules and functions and, if syntactically invalid rules or functions are created, the language's validity checker will identify them and make sure they are corrected prior to their use during a simulation.

7.3.3. Line Wrapping

As expected, line wrapping became a problem for the users. Given the complexity of their expressions and the length of their object.slot names, it was inevitable that they would have difficulty viewing and understanding expressions that could not be displayed in a single screen. In one example, I had to scroll the window four times to view the entire expression. Although I could have enlarged the window, the expression could still not be viewed without scrolling and, even if it could, there could easily be much longer expressions. While I did not have time to address this, another developer who is currently working on the project has just added functionality whereby expressions will be wrapped based on user preferences. For example, users can specify that lines will be wrapped after an "=" sign or at the end of each sub-expression. Although the users have not had a chance to use this new functionality yet, they are likely to find it quite useful since it will give them the ability to view their expressions with a minimum of scrolling.

7.3.4. Function Argument Lists

Function argument lists must be created and edited by hand. This is in contrast to the rest of the programming environment, which provides structure editors for most editing tasks. The only other exceptions are in-line editing and the editing of external functions. While neither of these were ever meant to make use of structure editors, the editing of function argument lists was meant to. I opted for free-form editing with post editing validity checking because I ran out of time to implement the structured approach. While the users would prefer to have the editing of argument lists guided by a

structure editor, they find the manual editing required to be a minor inconvenience. The syntax is straightforward and only the names of the argument types need to be remembered. Moreover, the names of the argument types are listed in a pop-up menu for the function's return type if the users need to look up the available options. The fact that the users were not particularly concerned with this came as a surprise to me; I had expected this to be a significant source of difficulty. Given the fact that the users are not overly concerned with this issue and given that amount of time that would likely be required to provide a structured editing solution, this functionality will most likely be postponed until after more important functionality is added. In fact, it is entirely possible that a structured editing solution will never be added.

7.3.5. Resizable Editors

Displaying only a subset of each editor's fields turned out to be only partially successful. Recall that all editors allowed the users to display either a subset of their fields or all of them, with the default being a subset. This works well in some cases. For example, hiding each ruleset element's description text can be convenient in that it allows the editor to consume less screen real estate. In other cases, however, the hidden fields need to be visible at all times. The main example of this is a rule's execution constraint. Since this is used to determine the conditions under which the rule can be executed, it needs to be visible at all times. Because this field is hidden by default, users sometimes do not realize it is there. As a result, rules writers have sometimes put the rule execution constraint in the rule body instead. They do this by changing a simple assignment statement into a conditional assignment statement or adding a condition to an existing conditional assignment statement. While this works, it has two potential drawbacks. The first is that it can make the policy more difficult to understand. For example, if there are a lot of assignment statements, each will have to use this condition and it may not be clear that this condition applies to the entire rule. The second is that it can lead to incorrect rules if the programmer fails to add this condition to each of the rule's assignment statements.

The fact that the resizable editors do not work as well as intended is disappointing given the amount of time it took to create them. As mentioned in the previous chapter, this demonstrates the need

to prioritize the tasks associated with a software development project and the importance of not allowing oneself to get caught up in the challenge of solving problems when there is not a demonstrated need for the solutions that they provide. In hindsight, the functionality provided by the resizable editors could have easily be accomplished using other, less development intensive means.

7.3.6. Debugging

Terry expressed an interest in better debugging support. He said that the available tracing support does not fully meet their needs and makes the debugging of rules difficult. An example of this involves the displaying and manipulating of intermediate values calculated during a simulation. These values are generally returned by a function and are used in the calculation of the final values that are assigned to object.slots by the rules. As the language and environment work now, the intended method for users to save and view these values is through the tracing facility. While this capability is helpful, it falls short of what the users really need, which is to assign these intermediate values to an object.slot so that they can use the object's data manipulation and display facilities to perform post-execution analysis on these values.

Given how the language is designed, however, it is not possible to directly set an intermediate value to an object.slot during a simulation. Values can only be set at the rule level using an assignment statement. As a result, users have two choices if they want to set intermediate values on an object.slot. The first is to log these values as trace statements during the simulation run, pull them from among all of the other tracing materials produced during a simulation run and then place them in an object.slot. This approach can be time-consuming and error prone. The second is to create an additional assignment statement as part of the rule that sets this intermediate value to an object.slot. This approach addresses the problems associated with the first, but has some of its own. The main one is that this intermediate value cannot be used by the rule during its current execution. This is because before a rule is executed, it takes a snapshot of the *RiverWare* model and uses this snapshot for all of its object.slot reads. This is to insure that the order of a rule's assignment statements do not affect the values that the rule produces. As a result, calculating intermediate values in this manner may not be of much help. In addition, this

solution would clutter up the rule with an assignment statement that is not part of the rule's policy, which could lead to confusion or at least distraction among those reading the rules. Lastly, this solution could negatively impact the simulation's performance by including an assignment statement in a rule that does not contribute to the mass balancing of any object in the model. If this assignment statement is the only one on the rule to set its value, the simulation engine would think the rule successfully executed when in fact it has only calculated an intermediate value. This would result in the simulation engine and rule processor passing control back and forth without actually calculating any results necessary to complete the simulation.

One possible means of addressing this problem would be to allow assignment statements within functions. While this would certainly address the problem, it would have a number of negative impacts. It would distribute the setting of values so that it would no longer be immediately clear which object.slot values a rule was setting and why. In addition, it would allow for the creation of functions with side-effects in that a function could calculate and return a value that was also set to an object.slot. Even though the value set to the object.slot would not be usable until the next execution of a rule, the side-effect would likely make the policy more difficult to understand. While the stated purpose for this functionality would be for post-execution analysis, I have no doubt that this would be used as a means of setting object.slot values for use in mass balance calculations.

An alternate approach would be to allow functions to write their debugging information to object.slots. This would be in addition to its current ability to write this information to *RiverWare's* tracing facility. In order to avoid the problems discussed above, the object.slots to which these data would be written would need to be write-only while the simulation was running. Once the simulation is complete, the object.slot could be treated as a normal object.slot and be either read from or written to. This would provide the requested functionality without opening the language to the kind of misuse it was designed to prohibit.

7.4. Conclusion

The new language and programming environment have been in use within *RiverWare* for over four years and are allowing the users to create simulations that would have been difficult or impossible using their older tools. While neither the language nor the programming environment are perfect, they combine a sufficient amount of accessibility and expressiveness to allow a wide range of users to create, update and understand the policies needed to model and control the river systems for which they are responsible.

Chapter 8

General Lessons

8.1. Introduction

This chapter brings together and summarizes the lessons learned during the two case studies. The intent of these lessons is to provide insight for designers of an application-specific language. Each lesson is accompanied by a brief clarifying statement and, where applicable, examples from the case studies. The lessons are organized as general lessons and lessons that relate to target users, problem descriptions, programming languages and programming environments. The chapter ends with some future directions for research in application-specific language design.

8.2. General

The primary goal of the design effort is to provide a solution that meets the users' needs and not necessarily design and develop a language. The creation of an application-specific language should only be undertaken if it provides the best solution to the users' problems. If a superior alternative exists that does not require the creation of a language, this option should be seriously considered. It should be selected if the effort to design and develop this alternative is less than the effort to design and develop a language. This is particularly true if the users have little to no programming experience.

The effort necessary to design and develop an application-specific language should not be underestimated. The time necessary to understand the users and their problems and to understand the various options available in the design of an application-specific language can be considerable. In addition, the development of an application-specific language can entail a significant amount of work, especially if the developer has never attempted this before.

Time and budget constraints will likely have a significant affect on the design effort. While it might be tempting to overlook these, they are a reality for most programming projects. Both language design efforts, although particularly the first, were influenced by limited time and budget. The selection of Tcl for the first language was largely predicated on the need to quickly produce a usable language.

An application-specific language should be designed to encourage and, where possible, enforce the creation of programs that are readable, maintainable and correct. Programming languages need to be designed in such a way that they address both the human and computer audiences for which they are intended. It should not be sufficient that the programs written in a language produce correct results. They also need to be accessible to the people who will need to read and maintain these programs.

Working within a framework in which target problems and target users are central is important in creating a language that meets the users' needs. Target problems, the focus of Problem-Centered Design [Lewis, Rieman and Bell, 1991], serve to narrow the design alternatives to those languages that can effectively represent the users' problems. Target users can further narrow the design alternatives to those languages that will be appropriate given their characteristics and abilities.

Design goals can help narrow the range of possible language options by focusing the design effort on the users' needs in addition to the problems that they need to express. Other than expressiveness and facility, design goals were not explicitly considered during the design of the first language. For the second language, however, readability, writability, maintainability, familiarity, safety, performance and backwards compatibility were goals that were used to drive the design of the programming language and environment.

It may be less costly to first design and develop a temporary language than to try to develop the final language up front. The time needed to design and develop a usable version of the first language was relatively brief. While the resulting language was not ideal for the users, it served to

make clear what they needed in the second language. An attempt to design and develop this second language, or one like it, outside of the context of the knowledge gained from the first language would have required significantly more time and effort. Given that it was not clear what the users actually needed until after the completion of the first language and the observation of its use, it is unlikely that this language would have met the users' needs. By performing an initial design and development cycle, a relatively small amount of time and effort was spent gaining the knowledge needed for the design and development of the final language.

If an initial language is to be created to gain an understanding of the users, the designer should be careful not to spend an excessive amount of time refining it. The purpose of an initial language is to gain insight into the needs of the users. It should be complete enough to allow the users to create and use their programs in the context of the application. Any additional development time, however, may lessen the time available to design and develop the second, and hopefully final, language. During the first case study, too much time was spent refining the language and environment, which delayed the creation of the second language.

An initial language can provide the users with the ability to communicate what they need in an application-specific language system. Users' may be unable to convey their needs without the ability to use a language in the context of an application. The use of an initial language can help the users understand how a language will interact with the application, what types of problems it can solve and how it can solve them. Prior to the creation of the first language system, the users had difficulty expressing their needs. Use of the first language, however, allowed them to see both its power and its limitations and, as a result, helped them provide critical and useful feedback regarding their needs.

Clearly separate *nice to have* functionality from necessary functionality. The time needed to design and develop optional functionality will likely reduce the time available for the design and development of other, more important, functionality. Accordingly, each piece of functionality should be prioritized and its design and development time estimated. If it appears that too much effort is required

for a relatively unimportant piece of functionality, an alternative, including omitting the functionality, should be selected.

An application-specific language system will entail more than simply a programming language. In addition to the language, a subroutine library, a programming environment and a language processor may be required. A subroutine library can provide functionality that cannot be easily included in the language, a programming environment can provide users with the support needed to load, save, view, edit and debug programs, and a language processor can control the execution of the programs and allow the application and the programs to communicate.

Avoid the use of problem-specific logic in the language processor. Placing problem-specific logic in the language processor, as was done with the *RSS* rule language, can restrict the range of problems that can be represented in the language. Keeping the language processor problem-neutral will maximize the number and type of problems that can be solved using the language system, which is important given that the initial problems supplied by the users are unlikely to constitute the entire set of problems that they will eventually want to solve.

If user tests are performed, have the users 'think aloud' [Lewis, 1982] in order to gain an understanding of what they find obvious and what they find nonintuitive. Users generally need to be encouraged to verbalize their thoughts as they use a system. Their thoughts can make clear the intent of actions and the meaning of any periods of inaction. This was done for both user tests and helped me gain an understanding of how the users perceived the system and made clear many of the difficulties that they were having.

8.3. Target Users

The users' primary goal may be to create programs that produce results that *they feel* are correct. Do not assume that users, especially novice users, will be interested in using the language in the manner that trained programmers are likely to use it. Users may be more interested in having their programs produce results that they believe to be correct than in creating correct, readable or

maintainable code. This was noticed in the misuse of dependencies and global variables during the first case study.

Seek a diverse set of target users. Involve as varied a group of target users as possible as early as possible. Get input from these users in the form of target problems in the early stages of the design and in the form of programming walkthroughs and user tests as the language matures. I failed to involve the CRMUIG users until late in the development of the first language and, as a result, missed their need for a simpler, more readable language.

Inexperienced users may benefit from a language that limits flexibility and provides significant programming support. Languages that limit flexibility can encourage users to create readable, maintainable or correct programs. A supportive programming environment can offset this lack of flexibility by making it easier for the users to write and update their programs. The first language was very flexible and included a programming environment that provided them with little support. The resulting rules were relatively easy to write, but were difficult to read and maintain and were often incorrect. The second language placed considerable limitations on the creation of rules, yet provided a great deal of programming environment support. The resulting rules were also relatively easy to write, but were easier to read, maintain and less prone to errors.

User preferences are a double-edged sword. User preferences can make clear the languages and language types that users are likely to embrace and rapidly learn. Yet, they can potentially bias users toward languages that do not fit their experience and abilities. The users were familiar with procedural programming languages and favored those for the first language. While some were able to effectively use the first language, many were not. The second language was functional and not an obvious choice for the users, yet it met their needs better than the first language.

The users' desire to program can have a significant affect on the quality of the code produced. If programming is a secondary part of the users' jobs and they do not consider themselves programmers or care to become better programmers, they are likely to be more concerned with

producing correct results than producing code that can be read and maintained by others. For these users, it is important to give them a language that will help them produce readable and maintainable code since they are unlikely to produce it on their own.

The users' reason for using the language can influence the latitude available in the design. If the users are required to use the language, the designer may be able to push the design of the language in a direction that better meets the users' needs. If the users are not required to use the language, its use must provide appreciable benefits without being unduly cumbersome. Users will seek other solutions if the cost of using the language is too high.

The frequency with which the language is used and the manner in which programs are shared should influence the language design. If the language is to be used mostly by individuals who are primarily tasked with programming, a premium can be placed on the making the language expressive and writable. If the language is to be used sporadically or infrequently or the programs written in it are to be shared or read by others, a premium should be placed on making the language readable and maintainable.

8.4. Problem Descriptions

Attempt to get problem descriptions from eventual users of the language. Users may include programmers and those who will need to read and understand the programs written in the language. Although the designer can create problems descriptions, they are likely to be artificial and not reflect the true needs of the users.

Problem descriptions are likely to be incomplete and/or incorrect. The problem descriptions provided by the users will probably be neither complete nor correct. Although the users may have a high-level understanding of the problems that they need to solve, they may not be able to express the details outside of the context of the language's use within the application.

Problem descriptions may need to be disambiguated. Users may not be able to unambiguously express their problem descriptions without help for the language designer. It may be helpful to convert what the users provide into a temporary language that can be used as the basis for iterative disambiguation. While this may work, it is risky in that it can bias the design of the final language to one that is similar to this temporary language. The problem descriptions for the first language needed a significant amount of effort to make them unambiguous and could not be completely disambiguated until the users were provided with a language to be used within *RiverWare*.

Use the problem descriptions in the context of the users' characteristics and goals. Problem descriptions on their own can lead to any number of languages. Accordingly, the characteristics and goals of the users need to be taken into account in order to narrow the choice of languages and drive the design to meet their needs. The design of the second language was highly dependent on the users' characteristics and goals and resulted in a language that focused on readability, maintainability and safety.

Programming environment problem descriptions may be difficult for the users to provide. The users may only be able to provide instances of desired features rather than specific problems. To the extent possible, try to gain an understanding of the underlying problems that they are trying to solve and use these.

Problem descriptions for subroutines are likely to be part of the problem descriptions for the language. The users are unlikely to explicitly provide problem descriptions for subroutines. The degree to which the users' problems are addressed with subroutines will largely depend on the base language and the designer's preferences.

Pay attention to the form of the problem descriptions provided by the users. The form of the users' problem descriptions can provide insight into what representation of their problems they find intuitive. This form can, in turn, be used as a model for a language that the users find similarly intuitive. The first language did not closely follow the form of the original problem descriptions and was difficult

for the users to use. The second language more closely followed the form of the original problem descriptions and was easier for the users to use.

8.5. Programming Language

If a language can be misused, it probably will be. Programmers, especially inexperienced ones, are likely to be more concerned with completing their task than in producing elegant code. As a result, they will put effort into making their programs produce results that meet their expectations and may not worry about writing programs that are readable, maintainable and even correct. The first language allowed the users to misuse dependencies and global variables.

A general purpose language can be beneficial in cases where the problem descriptions are ill-defined and the cost of future language changes is high. A general purpose programming language may, with the addition of application-specific functionality, be usable as the basis for an application-specific language. The resulting language is likely to be highly expressive and flexible and, therefore, able to be used to represent future problems without requiring significant changes. The first language was both highly expressive and relatively easy to update.

A general purpose programming language may not be a good choice for novice programmers who do not intend on becoming programmers. General purpose programming languages can provide more flexibility than needed and allow the users to create unreadable and unmaintainable programs. This is in part due to the fact that general purpose programming languages are intended to be used to represent a large range of problems and will not necessarily provide a close mapping to the users' problem domain.

A task-specific language is likely to be better for non- and novice programmers than a general purpose language. A highly structured language is likely to be a good choice for novice programmers as the added flexibility associated with an unstructured language may not match their abilities and propensities. The less flexibility a language has and the more closely it maps to the users'

problem domain, the less likely the users will get into trouble by making inappropriate use of a language's flexibility.

Designing a custom language will maximize flexibility, but will entail considerable design and development effort. Given the complexity of language design, borrowing a language should be considered first and rejected only if the resulting language does not meet the users' needs. The time needed to design and develop a custom language can be considerable, especially if the designer is not experienced in the design and development of programming languages.

The use of an existing language can speed the design and development of the language, but will provide less flexibility in terms of language design. Existing languages, such as Tcl, Perl, Python and Java, can speed the implementation of an application-specific language but will constraint the designer to the form of the chosen language. This may be an appropriate option if the language designer is inexperienced or has little time available to develop a language. It may be inappropriate if the resulting language cannot be effectively used.

The use of an existing language will require the creation of subroutines to tie the language to the application. At a minimum, subroutines will be needed to allow programs to read values from and write values to the application. They may also be needed to allow programs to execute application-specific functions. The first language, which was based on Tcl, relied heavily on predefined subroutines to pass information between its programs and the application. The second language, which was a custom language, built this same functionality directly into the language.

If performance is important and the language is not a simple language, it may be wise to use an existing language rather than design a custom one. Designing and developing a language that is both fast and expressive may prove to be difficult for an inexperienced language designer. Accordingly, an existing language may be preferable and allow the designer to focus on other aspects of the language that the users feel are important.

Integrate features into language as opposed to using subroutines. If a concept can be easily and naturally included in the language, it should be as this is likely to make the representation of the concept more straightforward. The syntax needed to access state variables (i.e., object.slot values) was included in the second language and resulted in programs that were easier to write and read than their counterparts in the first language.

The language should be structured so that important parts of the programs are obvious. Users will not always structure their code in a manner that highlights the important parts of a program. Where possible, the language should enforce this, since not doing so can lead to difficult to read and maintain programs. The second language did this in two places. The first was to make functions central to the language so that the concepts that were part of policies could be easily and succinctly expressed. The second was to require the setting of values at the topmost level of the rules so that what was being set and why it was being set would be obvious.

Frequently used functionality should be codified and encapsulated. Frequently used programming constructs should be encapsulated and, where possible, directly supported by the language. Examples in the second language include rule execution constraints and function post execution constraints. While these constructs were not strictly necessary in that the language supported the underlying functionality, their inclusion made it clear when they were needed and under what conditions they applied.

Do not require users to know the underlying form of a data type. Users should be able to access and manipulate data in their programs without being required to know the manner in which the data is represented. This allows the user to think about the data in the abstract and allows the designer to change the underlying form of the data type, if needed. Support routines, like those included for the date/time routines in the first language, might be necessary.

The inclusion of global variables can make the creation of programs easier, yet lead to programs that are difficult to read and maintain. The inclusion of global variables can allow users to use poor programming practices, such as the creation of functions with side-effects. This happened during the first case study and, while the creation of the rule was accelerated, subsequent maintenance was difficult. In addition, it was difficult to understand the rule's underlying policy.

A language that supports values with different scales and units should verify that all comparisons and combinations of these values are valid. Values with different scales and units cannot always be combined and compared. In some cases, the values need to be converted to compatible scales and units prior to being combined or compared. In other cases, their combination and comparison is invalid and should not be allowed. In either case, a language that does not at least check that value comparisons or conversions are valid is unsafe. When compatible values are combined, the language should automatically do so and assign the correct scale and units to the resulting value.

A safe and slow language is generally better than a fast and unsafe language. A unsafe language can lead to incorrect results, which are unlikely to be useful regardless of the speed with which the programs execute. Examples for the case studies include automatic dependency determination and unit checking and conversion. Both of these could be misused in the first language, which led to programs that generated incorrect results. Neither could be misused in the second language, although this entailed increased development and execution time.

Documentation should be provided in order to aid users as they learn the language and to help them to become self-sufficient. Written documentation can include user guides, example-based cookbooks and reference manuals. Training can be instructor led or self-directed. Regardless of the particulars, this up-front documentation and training can facilitate the acceptance and use of the language. Documentation for both languages was inadequate and should have been a higher priority.

The wording used in error messages should be accessible to those who will read and interpret them. Error messages need to written so that they can be understood by the language's users. Accordingly, technical jargon should be avoided for a language that is to be used by inexperienced programmers. As with documentation, neither language did a particularly good job of making error messages accessible to the users.

The time necessary to create accessible error messages needs to be built into the schedule as it will likely be overlooked otherwise. Error messages are not explicitly part of the language and are likely to be added while the language is being developed and debugged. Accordingly, they may be worded for the benefit of the programmer rather than the end user and will need to be reworded prior to the release of the language. If time is not allocated for the rewording of the error messages, it is unlikely to be done.

The omission of an optional part of the language should not lead to incorrect results. Users should not be able to create programs that generate incorrect results by leaving out optional parts of the language. The language should be safe, even if this leads to decreased performance. In the first language, rules' dependencies were optional, yet their omission could and did lead to incorrect results.

Do not require the users to specify more information than they can reasonably get from their programs. The first language required the users to determine the inclusive list of dependencies for each rule. Given the complexity of their rules, this was difficult to do and, as a result, the users often did so incorrectly, which led to poor performance and incorrect results.

If programs will be used with multiple sets of application data (e.g., model files), they should be stored in their own files and loaded separately from the rest of the application data. Storing programs in their own files will facilitate the sharing of programs among different sets of application data. It will also require separate load and save functionality.

The storage of the programs will affect how the elements of the program are shared. All elements related to a set of programs can be stored in a single file or multiple files. If they are stored in a

single file, the users will have to explicitly share them. If they are stored in separate files, they can be shared by reference. The option selected will depend on the users' needs. If there will need to be a significant amount of sharing of these elements, multiple files are warranted. If not, a single file will likely suffice. If a single file in the context of a structured editing environment is used, copy and paste operations will be needed to support sharing.

Be cautious of creating new control structures for the language. The commonly used procedural control structures work well and it is unlikely that there are new ones or variations on existing ones that will be an improvement over what is available. In an effort to avoid the use of recursion, the second language made use of new *for* and *while* control structures. While they worked as intended, the users found them nonintuitive and difficult to use.

With new or unfamiliar language features, make sure that the intent of the whole and the pieces is obvious. Users will generally know how to use common language features. If new language features are included, the language or programming environment should be structured in order to make it obvious how they should be used. The *for* and *while* control structures used as part of the second language were unfamiliar and would have benefited from a clearer syntax.

The data types supported by the application can be used to determine which data types will be needed in the language. The application may not refer to them as data types, although the entities that are named and can be manipulated will likely need to be supported in the language. If debugging is to be allowed, a string type should also be included even if it is not supported in the application.

A placeholder data type will be necessary if the programs are required to be syntactically valid at all times. The use of a structure editor, for example, will require the use of the 'unknown' data type to allow incomplete programs to be saved, loaded, viewed and edited while still maintaining the syntactic correctness of the programs.

If programs are to be used to convey information to non-programmers, they should be understandable outside of the context of their execution. Non-programming users may not have access to the application and, even if they do, might not be able to understand how the logic of the programs is affected by their execution. Some language features, such as global variables, which can be used to create functions with side-effects, can lead to programs that cannot be understood outside of the context of their execution and should be excluded from the language unless there is a compelling reason to include them.

Backward compatibility should be supported for languages that have existing programs that will need to be used with the new language. In cases where an old language is to be replaced, users should not be expected to immediately rewrite their existing programs. The new language should support the programs written in the old language or a means of converting the existing programs to the new language should be provided.

8.6. Programming Environment

The programming environment can significantly impact the usability of the language. The importance of a programming environment should not be underestimated. A good one can significantly improve both the readability and writability of programs created in the language.

Where possible, the look and feel of the programming environment should match that of the application. This will provide the users with a more consistent user experience and require less context switching when moving between the application and the language's programming environment.

Use consistent user interface conventions in the programming environment. Names, available actions and appearances should be consistent throughout the programming environment. To this end, it will probably be wise to step back during its design and development to ensure that any inconsistencies are removed. In the second programming environment, the use of single and double clicks to select, toggle, edit and open language elements was not checked until after development was complete and, as a result, was inconsistent and occasionally confusing.

Options for displaying programs include unformatted text, syntax highlighted text, a mixture of text and graphics and graphics. Unformatted and syntax highlighted text will allow the designer to make use of existing editors. This can be particularly useful if the time to design and develop the programming environment is limited. The use of graphics, either with or without text, will require the creation of a custom editor, which can be used to provide the user with a considerable amount of programming support. The second programming environment included a structure editor to assist the users in the reading, writing and maintenance of their rules.

Display all critical program logic by default. If the programs editors support the hiding of information, the hidden information should only be that which does not affect the program's results. Examples of information that can be hidden include comments, formatting preferences and debugging-related information and commands. The rule editor for the second language hid rule execution constraints by default, which caused the users to move the logic intended for them into the body of the rule. While doing so does not necessarily alter the manner in which the rules execute, it can obscure the meaning of the rule's policy and can lead to incorrect results.

Allow users to perform all programming-related tasks within the programming environment. Do not require the users to leave the programming environment in order to perform a task that is integral to the creation, editing or analysis of the language's programs. An example of where this principle was violated was in the first language in which the users were required to use the host operating system to create ruleset-related files.

Provide an obvious means of cancelling an action. If an action is supported, there should be a clear means of cancelling it. A counter example to this was in-line editing that was used in the rule and function editors. Once invoked, the user was required to perform an undocumented and nonintuitive action to cancel the editing session.

If the result of an editing action is checked for correctness and found to be incorrect, the user should be given the option of cancelling the editing action or re-editing the incorrect result. Cancelling the editing action allows the user to return to the pre-edited text. Re-editing the incorrect result allows the user to correct mistakes made during editing. In-line editing only allowed the user to return to the pre-edited text. As a result, the user was required to completely reenter any text that was not syntactically correct, which could be annoying if the error was small.

The programming environment should not force users to make premature programming decisions. Users should be able to leave parts of their programs unspecified prior to execution. If they are required to make premature decisions, they may add code that will need to be removed or altered later and then forget to do so. The function editors for the second programming environment briefly required this by requiring all functions to have a valid body to be used within an expression. As a result, users would add temporary bodies so that they could use of the functions elsewhere. This was identified and corrected during the first user test.

An editor should maximize the ability of the user to view what is being edited. The rule and function editors did not support the wrapping of rules or expressions. As a result, users could not always view an entire rule or function without being required to scroll.

If the programming environment supports editing of programs, it should also support saving them. While this may seem obvious, it was missed in the programming environment for the first language.

If a custom editor is provided, the programming environment should allow multiple editors to be simultaneously open so that elements of these programs can be shared and compared. Allowing for multiple editors to be open simultaneously will allow users to copy and paste program elements between programs. It will also allow two programs to be displayed using the same form and thus facilitate visual comparisons of the programs' contents. The second programming environment explicitly supported this capability.

If multiple editors from different programs can be simultaneously open, a visual grouping mechanism should be employed. Users should not be expected to examine the contents of the editors in order to determine the program to which they belong. Examples of visual grouping mechanisms include colors and patterns that are incorporated into the editors. The use of title bars, while helpful, is not as effective since it requires the user to read multiple title bars. The editors for the second programming environment were color coded to make clear the rulesets to which they belonged.

A structured editing environment can be used to aid readability by formatting the programs written by the users. A single, readable standard for displaying programs can be selected in order to minimize the ambiguity in the meaning of the programs. While this is somewhat restrictive, it does not diminish the expressiveness of the programs that can be written. The second programming environment used structure editors to insure that all programs written maintained a consistent appearance.

A structured editing environment can be used to guide the user through the creation of their programs. The guidance provided can be in the form of assistance at various points in the programs or in the form of constraints on the choices available at various points in the programs. The former can be used to provide the user with optional choices during the creation of their programs. The latter can be used to insure that the programs are always syntactically valid. The structure editors for the second programming environment constrained the users' choices as their rules and functions were created and updated to insure that they were always syntactically valid.

A structured editing environment can constrain the creation of programs in a way that is unnatural to those used to free-form editors. A structure editor will require users to create their programs so that they are syntactically valid at all times. As a result, and unlike a free-form editor, the manner in which the users create their programs can be unfamiliar. For novice users, this may be acceptable, since they will can benefit from the assistance and structure. For experienced users, the structure may become burdensome.

Programs should be saved in a human readable/editable form in cases where a structured editing environment is used. Users should be able to create and edit programs without using the structure editor. This will allow them to make simple changes without being required to invoked the editor and will allow users who do not have access to the structure editor to edit their programs. To allow this, the programs need to be stored in a human readable form, as was the case for the second language.

A programming environment that provides support can help infrequent or sporadic users. User who do not regularly use a system may forget details about either the language or the environment and may benefit from an environment that helps them come up to speed when they return to programming.

Editor-based programming support can apply to the whole program or can be based on the cursor's location. The former allows for advice or feedback on the entire program. Examples include spelling and syntax checkers and formatters. The latter allows for context dependent advice and feedback. Examples include structure editors and word-specific spelling checkers and correctors.

Editor-based programming support can be active or passive. Active support offers advice or enforces restrictions without the explicit request of the user. Passive support offers advice or makes corrections at the request of the user. The structure editors used in the second programming environment provided active support by restricting the creation of rules and functions to only those that were syntactically valid. The second programming environment also provided passive support by allowing the users to request a validity check on any part of the ruleset.

If active programming support is provided, the user should be able to deactivate it. Active programming support can be beneficial to users who are new to the language or programming environment. Accordingly, it is reasonable to make this the default editing mode in cases where it is provided. Users who no longer need the support, however, will likely find it a nuisance and should

probably be given a means of deactivating it. The structure editors provided as part of the second programming environment did not allow for programming support to be deactivated.

Editor-based programming support can be either required or optional. Required programming support cannot be overridden; the user is obligated to incorporate the advice or adhere to the restrictions provided by the editor. Optional programming support can be ignored or overridden by the user. The structure editors for the second programming environment employed required support.

The programming environment will likely be required to support maintainability. Maintainability has two facets. One allows users to view, understand and update programs written by themselves or others. The other allows users to make large-scale changes to programs when needed. The former can be addressed with a language that is readable and writable; the latter will likely require an environment that supports, at a minimum, global search and replace.

A means of debugging programs should be provided. If the programs created by the users are anything other than trivial, the language should support some method for allowing them to debug their programs. This could be as simple as print statements, as was used in both the first and second languages, and as complex as a symbolic debugger.

Debugging should permit the display of all information associated with the programs. If a structured debugging approach is used, it should support the display of all available information. If only a subset of the available information can be displayed, the users are likely to have difficulty understanding what their programs are doing. The second language failed to do this properly. A function's input and output could be displayed, but not the values of variables used within the function.

The language's debugging capabilities should be integrated with the application's debugging facility. Combining the two will allow the interaction between the execution of application-specific logic and the execution of the program's logic to be observed. This will allow the users to understand how they influence each other while executing. *RiverWare* did not provide debugging while

the first language was in use. For the second language, however, the debugging facilities of the two were tightly integrated.

The wording used on dialogs should be accessible to users. As with error messages, technical jargon should be avoided in the programming environment. The palette for the second programming environment used wording that was more appropriate for trained programmers than novice programmers.

The form used to display error messages and notifications should be the same as that used to display the programs in the editor. Errors related to specific sections of source code should be displayed using the same form as is used when editing. If the editors display programs using structured text, the users may find it confusing to have errors displayed using plain text since this will require them to map the text in the error message to the structured text displayed in an editor. Error messages in the second language were displayed using plain text, rather than in the context of the structure editors.

8.7. Future Directions

How can the language designer get problem descriptions in sufficient detail if the users are not able to provide this information outside of the context of their use within the application? The users often know what they want at a high level, but are unable to provide detail sufficient to design a language without being able to create and use their programs in the context of the application. One possible solution is to create a partial implementation that will be used to get feedback on the design of the ultimate language. Time and resources may not be available to allow for this, however.

Is there an effective means of getting programming environment-related problem descriptions from the users? Given that the programming environment is a critical component of an application-specific language, it is important to get good problem descriptions. The users may have difficulty supplying problem descriptions and may only be able to suggest a subset of the needed

functionality. Perhaps having users participate in scenario-based walkthroughs would help in this respect.

If a functional language is selected, are there alternatives to recursion? Recursion is often a difficult concept for new programmers to learn. Alternatives to recursion might include variations on procedural control structures and fill-in-the-blank templates that hide the use of recursion.

References

Arnold, Gosling and Holmes, 2000

Ken Arnold, James Gosling and David Holmes, *The Java Programming Language, Third Edition*, Addison-Wesley Publishing Company, Inc., Reading, MA, 2000.

Allen, 1978

John Allen, *Anatomy of LISP*, McGraw-Hill, New York, 1978.

Baecker and Marcus, 1986

Ronald Baecker and Aaron Marcus, Design Principles for the Enhanced Presentation of Computer Program Source Text, *CHI'86 Proceedings*, April 1986, pp. 51-58.

Bahlke and Snelting, 1992

R. Bahlke and G. Snelting, Design and structure of a semantics-based programming environment, *International Journal of Man-Machine Studies*, November 1992, Vol. 37, No. 5, pp. 467-480.

Baum, 1993

David Baum, This OOPs Tool Builds Apps That Can Reason, *DATAMATION*, Cahners Publishing Company, February 1993.

Behrens, 1991

Jon S. Behrens, *River basin operation: an object-oriented, artificial intelligence approach*, Ph.D. Thesis, University of Colorado, 1991.

Behrens, 1992

Jon S. Behrens, Simulation of Reservoir Operations Using Smart Reservoirs, *Proceedings of the ASCE Eighth National Conference on Computing in Civil Engineering*, June, 1992, Dallas, TX, pp. 606-613.

Bell, et al., 1994

B. Bell, W. Citrin, C. Lewis, J. Rieman, R. Weaver, N. Wilde, and B. Zorn, Using the Programming Walkthrough to Aid in Programming Language Design, *Software–Practice and Experience*, Vol. 24 (1), John Wiley & Sons, Ltd., January 1994, pp. 1-25.

Bellamy and Carroll, 1992

Rachel K. E. Bellamy and John M. Carroll, Re-structuring the programmer's task, *International Journal of Man-Machine Studies*, November 1992, Vol. 37, No. 5, pp. 503-528.

Böcker, Fischer and Nieper, 1986

Heinz-Dieter Böcker, Gerhard Fischer and Helga Nieper, *Proceedings of the ACM SIGCHI '86 Conference, ACM*, April 1986, pp. 44-50.

Booch, 1986

G. Booch, *Software Engineering with Ada, Second Edition*, The Benjamin Cummings Publishing Company, Inc., Menlo Park, CA, 1986.

Brownston, et al., 1985

Lee Brownston, Robert Farrell, Elaine Kant and Nancy Martin, *Programming Expert Systems in OPS5*, Addison-Wesley Publishing Company, Inc., Reading MA, 1985.

Brooks, 1982
Frederick P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1982.

Buchanan and Shortliffe, 1984
Bruce Buchanan and Edward Shortliffe (Eds.), *Rule-Based Expert Systems*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1984.

Buckley and Wheatcraft, 1991
Brian Buckley and Louis Wheatcraft, *Spacecraft Simulations with a re-usable Smart Control System*, Interface & Control Systems and Barrios Technology, Inc., March 1991.

Buckley, Buckley and Chiang, 1976
John W. Buckley, Marlene H. Buckley and Hung-Fu Chiang, *Research Methodology & Business Decisions*, National Association of Accountants, NY and The Society of Industrial Accountants of Canada, Ontario, 1976.

Chandra, Richards and Larus, 1996
Satish Chandra, Brad Richards and James R. Larus, Teapot: Language Support for Writing Memory Coherence Protocols, *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996, pp. 237-248.

CLIPS, 1994
CLIPS Reference Manual, Software Technology Branch, Lyndon B. Johnson Space Center, January 26, 1994.

Cornell and Horstmann, 1996
Cary Cornell and Cay S. Horstmann, *Core Java*, SunSoft Press, Mountain View, CA, 1996.

Crivelli, 1995
Silvia Crivelli, *A Programming Paradigm and Library for Distributed-Memory Computers*, Ph.D. Dissertation, University of Colorado, Boulder, 1995.

Derby, Schnabel and Zorn, 1995
Thomas Derby, Robert Schnabel and Benjamin Zorn, A New Language Design for Prototyping Numerical Computation, *Scientific Programming*, John Wiley & Sons, New York, 1995.

Dorsey and Koletzke, 1995
Paul Dorsey and Peter Koletzke, *Oracle Developer 3 Handbook*, McGraw-Hill Professional Publishing, New York, 2001.

Eisenhardt, 1995
Kathleen M. Eisenhardt, Building Theories From Case Study Research, *Longitudinal Field Research Methods*, George P. Huber and Andrew H. Van de Ven (Eds.), Sage Publications, Thousand Oaks, CA, 1995, pp. 65-90.

Elshoff and Marcotty, 1982
James L. Elshoff and Michael Marcotty, Improving Computer Program Readability to Aid Modification, *Communications of the ACM*, May 1982, Vol. 25, No. 8, pp. 512-521.

Forgy, 1994
Charles L. Forgy, *Technical Overview of RAL*, Production Systems Technology, Inc., 1994.

Fourer, Gay and Kernighan, 1993
Robert Fourer, David M. Gay, Brian W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, The Scientific Press, South San Francisco, CA, 1993.

Fulp and Harkins, 2001

T. Fulp and J. Harkins, Policy Analysis Using RiverWare: Colorado River Interim Surplus Guidelines, *Proceedings of the ASCE World Water & Environmental Resource Congress*, Orlando, FL, May 2001.

Giarratano, 1993

Joseph C. Giarratano, *CLIPS User's Guide*, Software Technology Branch, Lyndon B. Johnson Space Center, May 28, 1993.

Göttler, 1992

Herbert Göttler, Diagram editors = graphs + attributes + graph grammars, *International Journal of Man-Machine Studies*, November 1992, Vol. 37, No. 5, pp. 481-502.

Green, 1989

T. R. G. Green, Cognitive Dimensions of Notations, *People and Computers V: Proceedings of the Fifth Conference of the British Computer Society*, R. Winder and A. Sutcliffe (Eds.), Cambridge University Press, 1989.

Green, 1995

Thomas Green, Noddy's Guide to Visual Programming, *Interfaces*, Autumn 1995, pp. 1-6.

Green and Navarro, 1995

T. R. G. Green and R. Navarro, Programming Plans, Imagery and Visual Programming, *INTERACT '95 Conference on Computer-Human Interaction*, Chapman and Hall, London, pp. 139-144.

Green, Payne and van der Veer, 1983

T. R. G. Green, S. J. Payne, and G. C. van der Veer, (Eds.) *The Psychology of Computer Use*, Academic Press, London, 1983.

Green and Petre, 1992

T. R. G. Green and M. Petre, When Visual Programs are Harder to Read than Textual Programs, *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics)*, CUD: Rome, 1992.

Green and Petre, 1996

T. R. G. Green and M. Petre, Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework, *Journal of Visual Languages and Computing*, 1996.

Gilmore and Green, 1988

D. J. Gilmore and T. R. G. Green. Programming plans and programming expertise, *The Quarterly Journal of Experimental Psychology*, Vol. 40A, No. 3, pp. 423-442.

Halewood and Woodward, 1988

K. Halewood and M. R. Woodward, NSEDIT: A Syntax-directed Editor and Testing based on Nassi-Shneiderman Charts, *Software–Practice and Experience*, October 1988, Vol. 18, No. 10, pp. 987-998.

Harbison, 1990

Sam Harbison, Modula-3, *BYTE*, November 1990, pp. 385-392.

Harland, 1984

David M. Harland, *Polymorphic Programming Languages*, Halsted Press, Chichester, West Sussex, England, 1984.

Hayes-Roth, Waterman and Lenat, 1983
Frederick Hayes-Roth, Donald A. Waterman and Douglas B. Lenat (Eds.), *Building Expert Systems*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1983.

Hilfinger, 1983
Paul H. Hilfinger, *Abstraction Mechanisms and Language Desig*n, The MIT Press, Cambridge, MA, 1983.

Hoare, 1980
C.A.R. Hoare, Hints on Programming Language Design, *Tutorial Programming Language Design*, A. I. Wasserman (Ed.), IEEE, Inc., NY, 1980, pp. 43-52.

Holt, et. al., 1988
Richard C. Holt, Philip A. Matthews, J. Alan Rosselet and James R. Cordy, *The Turing Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.

Horowitz, 1983
Ellis Horowitz, *Programming Languages: A Grand Tour*, Computer Science Press, Rockville, MD, 1983.

Hudak and Fasel, 1992
Paul Hudak and Joseph H. Fasel, A Gentle Introduction to Haskell, *ACM SIGPLAN Notices*, May 1992, Vol. 27, No. 5.

Hudson, 1994
Scott Hudson, User Interface Specification Using an Enhanced Spreadsheet Model, *ACM Transactions on Graphics*, July 1994, Vol. 13, No. 3, pp. 209-239.

Hudson, 1991
Scott Hudson, Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update, *ACM Transactions on Programming Languages and Systems*, July 1991, Vol. 13, No. 3, pp. 315-341.

Jørgensen, 1980
Anker Helms Jørgensen, A Methodology for Measuring the Readability and Modifiability of Computer Programs, *BIT*, 1980, Vol. 20, No. 4, pp. 394-405.

Kellogg, 1987
Wendy A. Kellogg, Conceptual Consistency in the User Interface: Effects on User Performance, *Human-Computer Interaction – INTERACT '87*, Elsevier Science Publishers, 1987, pp. 389-394.

Kohler, Poletto and Montgomery, 1999
Eddie Kohler, Massimiliano Poletto and David R. Montgomery, Evolving Software with an Application-Specific Language, *Workshop Record of WCSSS '99: The 2nd ACM SiGPLAN Workshop on Compiler Support for Systems Software*, Atlanta, Georgia, May 1999, pp. 94-102.

Leigh and Huffman, 1987
William Leigh and G. David Huffman, *Structured Fortran '77*, Mitchell Publishing, Inc., Santa Cruz, CA, 1987.

Leonard-Barton, 1995
Dorothy Leonard-Barton, A Dual Methodology for Case Studies, *Longitudinal Field Research Methods* George P. Huber and Andrew H. Van de Ven (Eds.), Sage Publications, Thousand Oaks, CA, 1995, pp. 38-64.

Lerner, 1992

Barbara Staudt Lerner, Automated customization of structure editors, *International Journal of Man-Machine Studies*, November 1992, Vol. 37, No. 5, pp. 529-563.

Lewis, Rieman and Bell, 1991

C. Lewis, J. Rieman and B. Bell, Problem-Centered Design for Expressiveness and Facility in a Graphical Programming System, *Human-Computer Interaction*, Lawrence Erlbaum Associates, Inc., 1991, Volume 6, pp. 319-355.

Lewis, 1982

Clayton Lewis, Using the 'thinking-aloud' method in cognitive interface design, IBM Research Report RC9265, IBM T.J. Watson Center, New York, Yorktown Heights, 1982.

Lewis, 1995

Simon Lewis, *The Art and Science of Smalltalk*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1995.

Lutz, 1996

Mark Lutz, *Programming Python*, O'Reilly & Associates, Sebastopol, CA, 1996.

MacLennan, 1983

Bruce J. MacLennan, *Principles of Programming Languages: Design, Evaluation, and Implementation*, Holt, Rinehart and Winston, New York, 1983.

Magee, Zagona and Frevert 2001

Timothy Magee, Edith A. Zagona and Donald Frevert, Operational Policy Expression and Analysis in the RiverWare Modeling Tool, *Proceedings of the Environmental and Water Resources Institute's (EWRI's) World Water & Environmental Resource Congress*, Orlando, FL, May 2001.

Mars, 1999

ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf

Masoner, 1988

Michael Masoner, *An Audit of the Case Study Method*, Praeger Publishers, NY, 1988.

Mettrey, 1991

William Mettrey, A Comparative Evaluation of Expert System Tools, *Computer*, February 1991, Vol. 24, No. 2, pp. 19-31.

Minör, 1992

Sten Minör, Interacting with structure-oriented editors, *International Journal of Man-Machine Studies*, November 1992, Vol. 37, No. 5, pp. 399-419.

Mössenböck and Templ, 1989

Hanspeter Mössenböck and Josef Templ, Object Oberon – A Modest Object-Oriented Language, *Structured Programming*, 1989, Vol. 10, No. 4, pp. 199-207.

Mössenböck and Wirth, 1991

Hanspeter Mössenböck and Niklaus Wirth, The Programming Language Oberon-2, *Structured Programming*, 1991, Vol. 12, No. 4, pp. 179-196.

Nathanson, 1978

M.N. Nathanson, Updating the Hoover Dam Documents, U.S. Department of Interior, Bureau of Reclamation, Denver, CO, 1978.

Neal and Szwillus, 1992

Lisa Neal and Gerd Szwillus, Structure-based editors and environments, *International Journal of Man-Machine Studies*, November 1992, Vol. 37, No. 5, pp. 395-398.

Neches, Swartout and Moore, 1984

Robert Neches, William R. Swartout and Johanna Moore, Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of their Development, *Proceedings of the IEEE Workshop on Principles of Knowledge-based Systems*, December 1984, pp. 173-183.

Neiman and Martin, 1986

Dan Neiman and John Martin, Rule-Based Programming in OPS83, *AI Expert*, Miller Freeman Publications, April 1986.

Neuron, 1994

Neuron Data Elements: White Paper, Neuron Data Inc., 1994.

Oman and Cook, 1990

Paul W. Oman and Curtis R. Cook, Typographic Style is More than Cosmetic, *Communications of the ACM*, May 1990, Vol. 33, No. 5, pp. 506-520.

Ousterhout, 1994

John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1994.

Petre, 1995

Marian Petre, Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming, *Communications of the ACM*, June 1995, Vol. 38, No. 6, pp. 33-44.

Petre and Green, 1990

M. Petre and T. R. G. Green, Where to Draw the Line with Text: Some Claims by Logic Designers about Graphics in Notation, *INTERACT '90 Conference on Computer-Human Interaction*, Elsevier, 1990.

Petre and Price, 1990

Marian Price and Blaine A. Price, Why Computer Interfaces Are Not Like Paintings: the user as a deliberate user, (Technical Report) *Human Cognition Research Laboratory*, The Open University, GB, 1992.

Phillipps, 1988

Ian Phillipps, *The International Obfuscated C Code Contest: Least likely to compile successfully*, http://www.ioccc.org/winners.html#P, 1988.

Kappa, 1993

Kappa: ProTalk Language Reference, IntelliCorp, Inc., September 1993.

Pratt, 1984

Terrence W. Pratt, *Programming Languages: Design and Implementation*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

Reitsma, Sautins and Wehrend, 1993

Reitsma, R., A. Sautins and S. Wehrend, RSS: A Construction Kit for Visual Programming of River Basin Models, *Proceedings of Water Management in the '90s: a Time for Innovation*, Edited by K. Hon, 1993, pp. 264-268.

Repenning and Sumner, 1995
Alexander Repenning and Tamara Sumner, Agentsheets: A Medium for Creating Domain-Oriented Visual Languages, *Computer*, Vol. 28, No. 3, March 1995, pp. 17-25.

Roman, 1999
Steven Roman, Writing Excel Macros, O'Reilly & Associates, May 1999.

RSS, 1992
River Simulation System Technical Reference Manual, Center for Advanced Decision Support for Water and Environmental Systems, October 1992.

Schuster, 1987
Ronald J. Schuster, *Colorado River Simulation System - System Overview*, U.S. Department of Interior Bureau of Reclamation, May 1987.

Schwartz and Christiansen, 1997
Randal L. Schwartz and Tom Christiansen, *Learning Perl, 2nd Edition*, O'Reilly & Associates, Inc., Sebastopol, CA, 1997.

Selker, 1988
Ted Selker, Elements of Visual Language, *IBM Tech Report RC13929*, IBM T.J. Watson Center, New York, Yorktown Heights, 1988.

Selker and Appel, 1991
Ted Selker and Art Appel, Graphics as Visual Language, *IBM Tech Report RC16776*, IBM T.J. Watson Center, New York, Yorktown Heights, 1991.

Smedema, Medema and Boasson, 1983
C. H. Smedema, P. Medema and M. Boasson, *The Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

Smith, 1988
Jerry D. Smith, *An Introduction to Scheme*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.

Spinellis and Guruprasad, 1997
Diomidis Spinellis and V. Guruprasad, Lightweight Languages as Software Engineering Tools, *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997.

Steele, 1994
Guy L. Steele, *Principled Design of Programming Languages*, From presentation given June 22, 1994 while with Thinking Machines Corporation.

TclPro, 2001
TclPro, http://www.scriptics.com/software/tclpro/tclpro/TclProUsersGuide14.pdf, 2001.

Tennant, Ross and Thompson, 1983
Usable Natural Language Interfaces Through Menu-Based Natural Language Understanding, *Proceeding of the CHI'83 Conference on Human Factors in Computer Systems*, ACM, New York, pp. 154-160.

Unger and Smith, 1991
David Unger and Randall B. Smith, *SELF: The Power of Simplicity*, LISP and Symbolic Computation: An International Journal, 1991, Vol. 4, No. 3.

USBRa, 2002
http://www.usbr.gov/main/what/regionalmap/index.html

USBRb, 2002
http://www.usbr.gov/main/what/fact.html.

van Deursen and Klint, 1997
Arie van Deursen and Peter Klint, Little Languages: Little Maintenance?, *SEN Report R9704, Centrum voor Wiskende en Informatica*, Amsterdam, The Netherlands, March 1997.

van Laar, 1989
Darren Van Laar, Evaluating a Colour Coding Programming Support Tool, *Proceedings of the Fifth Conference of the British Computer Society*, Cambridge University Press, 1989, pp. 217-230.

van Rossum, 1995
Guido van Rossum, *Python Reference Manual*, Stichting Mathematisch Centrum, Amsterdam, 1991-1995.

van de Vanter, Graham and Ballance, 1992
Michael L. Van De Vanter, Susan L. Graham and Robert A. Ballance, Coherent user interfaces for language-based editing systems, *International Journal of Man-Machine Studies*, November 1992, Vol. 37, No. 5, pp. 431-466.

Welsh and Toleman, 1992
Jim Welsh and Mark Toleman, Conceptual issues in language-based editor design, *International Journal of Man-Machine Studies*, November 1992, Vol. 37, No. 5, pp. 419-430.

Wall, Christiansen and Schwartz, 1996
Larry Wall, Tom Christiansen and Randal L. Schwartz, *Programming Perl*, O'Reilly & Associates, Inc., Sebastopol, CA, 1996.

Wasserman, 1980
Anthony I. Wasserman (Ed.), *Tutorial Programming Language Design*, IEEE, Inc., NY, 1980.

Wehrend and Reitsma, 1995
Wehrend, S. and R. Reitsma, A Rule Language to Express Policy in a River Basin, *Proceedings of the ASCE Second Congress on Computing in Civil Engineering*, Edited by J.P. Mohsen, Atlanta, GA, June 1995.

Weverka and Poremsky, 2001
Peter Weverka and Diane Poremsky, *Word 2002: The Complete Reference*, McGraw-Hill Professional Publishing, New York, 2001.

Williams and Fisher, 1977
John H. Williams and David A. Fisher (Eds.), Design and Implementation of Programming Languages, *Proceedings of a DoD SPonsored Workshop*, Ithaca, October, 1976.

Williams and Hann, 1972
Jimmy R. Williams and Roy W. Hann, Hymo, A Problem-Oriented Computer Language for Building Hydrologic Models, *Water Resources Research*, February, 1972, Vol. 8, No. 1.

Wirth, 1974
N. Wirth, On the Design of Programming Languages, *International Federation of Information Processing*, 1974, pp. 386-393.

Wirth, 1984
Niklaus Wirth, History and Goals of Modula-2, *BYTE*, November 1984, pp. 145-152.

Wirth, 1990
N. Wirth, *From Modula to Oberon*, via ftp://ftp.inf.ethz.ch/pub/software/Oberon/Docu/ModToOberon.ps.gz, January, 1990.

Wolfram, 2001
Stephen Wolfram, *The Mathematica Book*, Cambridge University Press, Champaign, IL, March 1999.

Yin, 1994
Robert K. Yin, *Case Study Research, Second Edition*, Sage Publications, Thousand Oaks, CA, 1994.

Yourdon and Constantine, 1979
E. Yourdon, and L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

Zagona, et al., 2001
E. A. Zagona, T. J. Fulp, R. Shane, T. Magee and H. M. Goranflo, RiverWare: A Generalized Tool for Complex River Basin Modeling, *Journal of the American Water Resources Association*, August 2001, Vol. 37, No. 4.

Appendix A

Colorado River Basin Operating Rules

This appendix describes the current Colorado River Basin operating rules and related data as implemented in CRSM. As such, the implementation of these policies into the new model would be viewed as the absolute minimum requirement to replace CRSS.

Note that the headings do not reflect the rules as they ultimately were represented. For instance, "Reservoir Equation (Powell and Mead)" is listed a part of the "Reservoirs above Lake Powell" rule when it is in fact a separate rule.

A.1. Reservoirs above Lake Powell

The reservoirs above Lake Powell are operated by using monthly storage targets or rule curves. The rule curves are input for each reservoir, but will be changed by the model for Flaming Gorge, Navajo, Blue Mesa, Morrow Point, and Crystal, depending upon the elevation of Lake Powell. Four cases are checked:

1) If Lake Powell elevation is greater than 3690 feet (the turbine overload elevation), then the target content for each month is computed as follows:

for Flaming Gorge, Blue Mesa, and Navajo: target content = live capacity - target space

where the target space computation is detailed below.

for Morrow Point and Crystal: target content = live capacity

2) If Lake Powell elevation is less than or equal to 3690 feet and greater than 3570 feet (the rated power head), then the target content for each month is computed as follows:

for Blue Mesa, Navajo, Morrow Point and Crystal, the target contents are computed as for case (1).

for Flaming Gorge, target content is computed by:

target content = max {content at elevation 5946, computed content}

where computed content is based on the following equation:

computed content = live capacity of Flaming Gorge * storage ratio

The storage ratio is the sum of the current contents of Flaming Gorge, Blue Mesa, Morrow Point, Crystal, and Powell divided by the sum of the live capacities of those reservoirs. The elevation of 5946 at Flaming Gorge is the rated power head. It is noted that as the model runs, the content at that elevation does decrease due to sediment accumulation.

3) If Lake Powell is less than or equal to 3570 feet and greater than 3510 feet, then the target content for each month is computed as follows:

for Morrow Point and Crystal, the target contents are computed as for case (1).

for Flaming Gorge, Blue Mesa, and Navajo, target content is computed as in case (2), using the live capacity for each reservoir and the contents at 7438 feet for Blue Mesa (rated power-head elevation) and 6010 feet for Navajo.

3) If Lake Powell is less than 3510 feet, then the target contents for each month is computed as follows:

for Flaming Gorge, Blue Mesa, Navajo, Morrow Point and Crystal, the target content is set equal to the content at the minimum power elevation respectively. These contents are:

Reservoir	Contents
Flaming Gorge	5871 feet
Blue Mesa	7393 feet
Navajo	5990 feet
Morrow Point	7100 feet
Crystal	6700 feet

The computation of target space for Flaming Gorge, Blue Mesa and Navajo is accomplished as follows:

during the months of August through December, the target space (in 1000 ac-ft.) is “hard-wired” to be:

Reservoir	Aug	Sep	Oct	Nov	Dec
Flaming Gorge	100	150	210	280	350
Blue Mesa	25	50	100	150	200
Navajo	20	50	80	115	160

for the months January through July, the target space for each of these reservoirs is computed using a forecast. This forecast is computed as follows:

step a) compute the remaining “average virgin inflow” into the reservoir from the current month through July (from the following table, in 1000 ac-ft.):

Reservoir	Jan	Feb	Mar	Apr	May	Jun	Jul
Flaming Gorge	23.3	20.9	33.8	87.9	250.4	327.8	157.5
Blue Mesa	34.0	39.5	94.6	176.0	339.8	561.6	346.8
Navajo	18.8	24.6	69.3	176.9	297.3	284.7	120.1

step b) compute the remaining inflow into the reservoir from the current month through July (from the hydrology currently being used in the run)

step c) use a weighted average of the remaining inflow and the remaining average virgin inflow to

compute the predicted remaining inflow (from the current month through July):

Month	Inflow Weight	Average Virgin Inflow Weight
January	0.3	0.7
February	0.4	0.6
March	0.5	0.5
April	0.7	0.3
May	0.7	0.3
June	0.7	0.3
July	0.6	0.4

step d) compute the current month's release needed to store the predicted remaining inflow:

release needed for the current month = (current contents - live capacity + predicted remaining inflow) divided by the number of months remaining until the end of July

this release needed is checked to the minimum release and if it is less, it is set to the minimum.

step e) compute the maximum allowable storage in each reservoir that can still contain the predicted remaining inflow

for Flaming Gorge, where all values are for the current month:

max. allowable storage = content - release needed + inflow at Green River below Fontenelle + exports above Fontenelle + gains on Green River above Greendale, UT

for Blue Mesa, where all values are for the current month:

max. allowable storage = content - release needed + inflow at Taylor River above Taylor Park + gains on the Gunnison River above Blue Mesa

for Navajo, where all values are for the current month:

max. allowable storage = content - release needed + flow at San Juan River near Archuleta, NM - demand from Navajo Reservoir

each max. allowable storage is checked to the live capacity, and if greater, is set to the live capacity.

step f) finally compute the target space for the month:

target space = live capacity - max. allowable storage

Runoff Forecast Procedure - To simulate operations of Lake Powell and Lake Mead, the model needs to develop a runoff forecast. At Lake Powell, the runoff forecast is needed to avoid excessive releases at Glen Canyon Dam, and also to predict the EOWY contents of Lakes Powell and Mead to determine if equalization releases are necessary. At Lake Mead, the runoff forecast is needed to determine flood control releases at Hoover Dam during the January to July period.

Lake Powell Forecast - The model needs to develop a runoff forecast for Lake Powell, each month from January through September. Each month, the runoff volume for the current month being modeled through September needs to be forecasted. The use of a perfect forecast in determining the reservoir operation would unrealistically bias the results of a model run. So, a forecast error needs to be computed and added to or subtracted from a perfect forecast in order to more realistically simulate the reservoir operation.

The monthly Lake Powell runoff forecast can be computed by:

(1) The natural flow into Lake Powell for the current month being modeled through July,

plus

(2) The long-term average natural flow into Lake Powell for the months of August and September,

minus

(3) The estimated depletion in the Upper Basin during the forecast period,

plus

(4) A forecast error, which can be either positive or negative.

Forecast Error - The runoff forecast error is computed using equations derived from an analysis of past Colorado River forecasts and runoff data for the period 1947 to 1983. Analysis of these data reveals two strongly established patterns: (1) high runoff years are under-forecast, and low runoff years are over-forecast; (2) the error in this month's seasonal forecast is strongly correlated with the error in the preceding month's forecast. Since these established patterns are inherent in the equations used shown below,

A regression model was developed to aid in determining the error to be incorporated into the seasonal forecast for each month from January to June. The error is the sum of a deterministic and a random component. The deterministic component is computed from the regression equation. The random component is computed by multiplying the standard error of the regression equation by a random mean deviation selected from the standard normal distribution (mean = 0, variance = 1).

The forecast error equation has the following form (all runoff units are million acre-feet):

$$E_i = a_i X_i + b_i E_{(i-1)} + C_i + Z_r d_i$$

Where:

- i = month
- E_i = error in the forecast for month “i.”
- X_i = natural runoff into Lake Powell from month “i” through July.
- a_i = linear regression coefficient for X_i.
- $E_{(i-1)}$ = previous month's forecast error
- b_i = linear regression coefficient for $E_{(i-1)}$.
- C_i = constant term in regression equation for month “i.”
- Z_r = randomly determined mean deviation taken from the standard normal distribution.
- d_i = standard error of estimate for regression equation for month “i.”

The following table summarizes the regression equation coefficients for each month.

Month	i	Runoff Coefficient (a_i)	Error Coefficient (b_i)	Constant (C_i)	Standard Error of Estimate (d_i)
January	1	0.70	0.00	-8.195	1.270
February	2	0.00	0.80	-0.278	0.977
March	3	0.00	0.90	0.237	0.794
April	4	0.00	0.76	0.027	0.631
May	5	0.00	0.85	0.132	0.377
June	6	0.24	0.79	0.150	0.460

There are three additions to the procedure described above. (1) The magnitude of the June forecast error may not exceed 50 percent of the May forecast error. (2) The July forecast error is equal to 25 percent of the June forecast error. (3) There is no forecast error for the months of August and September.

Releases from Lake Powell (General) - The following is a description of how releases from Lake Powell are currently handled in CRSM:

Releases from Lake Powell are initially scheduled to follow an objective minimum release schedule, and then modified if necessary. Releases in excess of the minimum release schedule can be caused by either (1) Lake Powell exceeding maximum capacity or target storage levels, (2) releases dictated by the surplus strategy, if used, whereby excess releases are scheduled to avoid spills, or (3) releases needed to equalize the active contents of Lakes Powell and Mead by the end of the water year.

Objective Minimum Release - The CRSM schedules Lake Powell to maintain the minimum objective release of 8.23 million ac-ft. (MAF) from Lake Powell each year. This annual release is divided into 12 monthly releases which are specified in the CRSM input file. Releases from Lake Powell will be equal to these monthly releases unless one of the 3 conditions stated above occur.

602(a) storage requirements - The new model would have to have the same functionality as CRSM to compute the storage requirements in the Upper Basin so as to assure future deliveries to the Lower Basin without impairing annual consumptive uses in the Upper Basin. This storage is commonly referred to as "602(a) storage." As with all Policies and Constraints this would be written as a rule and modifiable by the user. Additionally 602(a) storage computations with other methods and data must be available to the user. Initially the new model will use the same method that is currently used by CRSM. This method uses the following four pieces of data:

1. The Length of the Critical Period
2. The Average Annual natural flow at Lee's Ferry for the critical period.
3. Percent shortage applied to Upper Basin scheduled depletions.
4. Amount of power pool to be preserved in Upper Basins reservoirs, if any.

The following describes how this data is currently used to compute 602(a) storage, however it should be noted that this method of determining 602(a) storage is not intended to be a formal determination of 602(a) storage as required by the *Criteria for Long-Range Operations of Colorado River Reservoirs.*

At the beginning of each calendar year a value for 602(a) storage is computed by the following formula:

602a = (UBD * (100 - PS/ 100) + MOR - NF + MPP

where:

602a is the 602(a) storage requirement

UBD is the sum of the expected Upper Basin Depletions over a period of "n" years where

"n" is equal to the length of the critical period. Current runs have been based on a critical period of 12, 25, or 48 years.

PS the percent shortage that will be applied to Upper Basin Depletions during the critical period, currently a value of 6.12 percent is being used. This value represents zero shortage of M&I demands and a 10 percent shortage on agricultural demands and so this value changes with the depletion schedule being used.

MOR is the sum of the objective minimum release for the next "n" years.

NF is the sum of the natural flow at Lee's Ferry for the next "n" years.

MPP is the amount of minimum power pool to be preserved in Upper Basin reservoirs, if any

Reservoir Equalization (Powell and Mead) - The equalization of storage between Lakes Powell and Mead will initially be simulated by the following steps. First the quantity of 602(a) storage needs to be computed, then the End of Water Year (EOWY) contents of Lake Powell, Lake Mead, and the sum of Upper Basin reservoirs is computed by the method described in the next section, then two checks are made to see if the reservoirs need to be equalized. If the sum of the contents of Upper Basin reservoirs is above required 602(a) storage, and if the predicted EOWY contents of Lake Powell are greater than the predicted EOWY contents of Lake Mead, then the active contents of Lake Powell and Lake Mead are equalized.

The volume of water required to be released from Lake Powell for equalization will need to be computed each month from January through September by taking half of the difference between the predicted EOWY contents of Lake Powell and Lake Mead and dividing by the number of months remaining through September. This procedure spreads the additional releases for equalization out over the water year and closely equalizes the two reservoirs by September 30.

Several constraints in the model will be needed to alter the amount of additional releases computed as described above. If the additional release would cause the total Upper Basin storage to drop below the 602(a) requirement, then the amount of additional release is reduced to prevent this from happening,. Likewise, the additional release is reduced if it would cause Lake Mead contents to exceed Lake Mead live capacity below exclusive flood control space.

The total release from Lake Powell is constrained by the maximum release. For Lake Powell, the maximum release is normally set to power plant capacity. If the additional release is reduced because of the maximum power plant capacity limitation, a greater amount will be released for equalization in subsequent months since the reservoirs will be further out of balance, unless of course the maximum release is also reached each subsequent month in the water year.

A different than normal pattern will develop if additional releases are being made for equalization and then the spring runoff is significantly less than the runoff used to predict the EOWY contents. In this case, in the month when the predicted EOWY contents of Lake Powell drops below the predicted EOWY contents of Lake Mead, the model will need to reduce the monthly objective minimum releases from Lake Powell to less than the monthly objective minimum releases in order to not exceed the annual objective minimum release. What this means is that in order to meet the annual objective minimum release, if the spring runoff was lower than expected and the releases early in the year were therefore

greater than needed, then the releases later in the year will be less than that which would normally be needed to meet the objective minimum.

A.2. Predicting EOWY Contents

The prediction of EOWY contents for both Lakes Powell and Mead needs to begin in January and for each month through the end of the water year.

Lake Powell EOWY contents are computed each month by taking the current contents, adding estimated inflow, and subtracting the assumed release, estimate of evaporation, and change in bank storage. Lake Powell inflow is estimated by taking the remaining monthly inflows from the hydrology record and applying a forecast error term composed of both random and deterministic components (this procedure is further described in the Lake Powell Forecasts Section). The model assumes that the Lake Powell releases for the remainder of the water year will follow the objective minimum release schedule.

Lake Mead EOWY contents need to be computed each month by taking the current contents, adding the Powell release (Mead inflow), and subtracting the Mead release, estimate of evaporation, and change in bank storage. The model would then compute Lake Mead releases for the remainder of the water year based upon downstream demands including any scheduled surplus releases.

Total Upper Basin predicted contents need to be computed by adding the current contents of Flaming Gorge, Blue Mesa, and Navajo Reservoirs to the predicted EOWY contents of Lake Powell.

A.3. Lake Mead Inflow Forecast

The operation of Mead for flood control requires a forecast in each month from January through July. For each month, the runoff volume for the current month through July is estimated. The flood control policy requires that the maximum forecast be used.

A mean forecast is first developed by taking the Powell current month through September forecast, subtracting the portion forecasted for August and September, and adding the long-term average natural tributary inflows between Lakes Powell and Mead for the current month through July.

The maximum forecast is then computed by adding a constant term to the mean forecast. These currently are:

Forecast Period	Constant Added (maf)
Jan - Jul	4.980
Feb - Jul	4.260
Mar - Jul	3.600
Apr - Jul	2.970
May - Jul	2.525
Jun - Jul	2.130
Jul	0.750

A.4. Lake Mead Flood Control

There are three flood control procedures currently used in CRSM in effect for different times of the year. The first procedure is used during the runoff forecast season (January through July). The objective during this period is to route the forecasted maximum inflow through the reservoir system using specific rates of Hoover Dam discharge, so that the reservoir is full by the end of July. The second procedure is used during the space building or draw-down period (August through December). The objective during this period is to gradually draw down the reservoir system to create space for the spring runoff. The third procedure is in effect throughout the year. Its objective is to maintain a minimum space of 1.5 million acre-feet in Lake Mead for rain events.

Procedure 1) Runoff forecast season (January through July)

We compute a value of minimum average release for Hoover using a mass balance:

a) assume Lake Powell will fill to 3,700 feet and Lake Mead will fill to 1219.61 feet (1.5 maf of space) by 1 August and compute the available space.

b) subtract the exclusive flood control space (1.5 million acre-feet)

c) add losses from Lakes Powell and Mead (evaporation, bank storage, and depletions) for the current month through July

d) subtract the maximum forecasted inflow for the current month through July

e) add the discharge from Mead assuming a Level i discharge, where i is the discharge level explained below.

If the computed release is greater than the assumed Level discharge, the release is recomputed using the next Level discharge. The iteration stops when the computed release is less than or equal to the assumed Level discharge and a final check is made. If the computed release is less than the next smaller level than the Level used in the computation, the release is set to that next smaller level.

The release computed by this method is the minimum average release for the month. Other criteria (such as downstream demand) will override this rule. However, the model never sets the release to a value less than this minimum. The release levels are given below.

Discharge Level	Release Rate (cfs)	Description
1	19,000	Parker Power plant capacity
2	28,000	Non-damaging release limit
3	35,000	Approx. Hoover Power plant capacity
4	40,000	Historic floodway capacity
5	73,000	Hoover controlled discharge capacity

Procedure 2) Space building or draw-down (August through December)

The available flood control storage space (storage below elevation 1229 feet, which is the “maximum

design flood pool") is specified as:

Date	Available Flood Control Space (maf)
1 Aug	1.50
1 Sep	2.27
1 Oct	3.04
1 Nov	3.81
1 Dec	4.58
1 Jan	5.35

However, these targets may be reduced to a minimum of 1.5 maf in each month if additional space is available upstream in active storage. The maximum storage space creditable for 1 Sep - 1 Jan is:

Reservoir	Maximum Creditable Storage Space (maf)
Powell	3.85
Navajo	1.0359
Blue Mesa	0.7485
Flaming Gorge & Fontenelle	1.5072

The model then sets Mead's rule curve to reflect the necessary space; however, these releases cannot exceed 28,000 cfs. Downstream demands can override this maximum, however.

Procedure 3) Exclusive Space Requirement (all year)

If available flood control space diminishes to less than 1.5 maf at any time during the year, then minimum flood control release are given by:

Mead Elevation (feet)	Available Storage (maf)	Releases (cfs)
1219.61 - 1221.40	1.500 - 1.218	release = inflow up to 28,000 cfs
1221.40 - 1226.90	1.218 - 0.340	outflow = inflow up to 40,000 cfs
1226.90 - 1229.00	0.340 - 0.0	outflow = inflow up to 65,000 cfs
> 1229.0		outflow = inflow

The model will force releases to maintain the space on a monthly basis.

A.5. Lower Basin Surplus/Shortage Strategy

There are currently two versions of CRSM, each with a distinct surplus strategy. The first surplus determination strategy is based on a percent probable assurance of avoidance of spills at Lake Mead. The second surplus determination strategy is based on an elevation or content trigger at Lake Mead.

The surplus strategy is the decision-making process used to determine when water in quantities greater than "normal" will be available for consumptive use in the Lower Basin and to distribute that surplus water to users. Users must have a contract with Reclamation to take any surplus water. As part of the Annual Operating Plan any surplus water is made available beginning October 1 and once made, is in effect for the remainder of that year and the following calendar year. Distribution of surplus water, however, may start on October 1 or January 1 depending on the particular user requesting the water.

This rule has three strategies, all of which would be considered a separate rules. These rules could be interchanged.

Strategy 1: The percent assurance of spill avoidance will be computed as

Step 0: fit the annual natural flow at Lee's Ferry to a normal distribution to get a relationship between percent exceedance and flow

Step 1: compute the total storage in Lakes Powell and Mead ($V_{tot,Oct1}$) at the beginning of the water year (October 1):

Step 2: compute the total storage allowed in Lakes Powell and Mead ($V_{tot,\ Sept30}$) at the end of the current water year (September 30)

$$V_{tot,Sep30} = V_{Powell} + V_{Mead} - 0.70 \times V_{fc}$$

where V_{Powell} is the maximum capacity of Lake Powell, V_{Mead} is the maximum capacity of Lake Mead and V_{fc} is the required flood control space for September 30. This assumes that 30 percent of the required flood control space is available in the other Upper Basin reservoirs. The required flood control space for September 30 is an input parameter and is currently set at 3.04 million acre-feet.

Step 3: compute the annual surplus release $Q_{surplus}$ (for the current water year) that may be required by Lake Mead

$$Q_{surplus} = Q_{\%avoid} - (CU_{UB} + CU_{LB}) - (V_{tot,Sep30} - V_{tot,OCt1})(1.0 + B_c)$$

where B_c is the user supplied bank storage coefficient for Powell and Mead combined (hard-wired as 0.0725), $Q_{\%avoid}$ is the natural flow at Lee's Ferry given the percent of avoidance input, CU_{UB} is the total scheduled consumptive use in the Upper Basin, and CU_{LB} is the total scheduled consumptive use in the Lower Basin.

• The total Upper Basin consumptive use is the total UB depletions plus the average evaporation in the Upper Basin reservoirs (user input as 560,000 acre-feet).

• The total Lower Basin consumptive use is the releases from Hoover + depletions above Hoover (Southern Nevada pumping from Lake Mead) + Mead evaporation (user input as 900,000 acre-feet) - annual average Glen to Hoover gains (user input as 801,000 acre-feet) - annual average gains and losses below Hoover (user input as 427,000 acre-feet) + Lake Mohave evaporation + Lake Havasu evaporation

where $E_{ann,moh}$ is the annual evaporation from Lake Mohave in feet, E_{month} is the amount of evaporation from a reservoir during a month in feet (user input), A_{moh} is the end of month surface area of Lake Mohave in acres. A similar equation is used to calculate the Lake Havasu evaporation.

$$E_{ann,moh} = E_{ann,moh} + E_{month} \times A_{moh}$$

Step 4: apply the following limits

• if the computed annual surplus release is negative or less than 200,000 acre-feet, then there is no surplus determined

• if the computed annual surplus release is between 200,000 and 300,000 acre-feet, then a surplus is determined and the total available is set to 300,000 acre-feet

• if the computed annual surplus release is between 300,000 to 13.4 million acre-feet, then a surplus is determined and the total available is the computed amount

• limit the total annual release from Hoover to 13.4 million acre-feet (approximately the power plant capacity of Parker Dam downstream): if the computed annual surplus release plus uses in the Lower Basin is greater than 13.4 million acre-feet, then set the computed annual surplus release to equal 13.4 million acre-feet - uses in the Lower Basin

Step 5: New diversion schedules are set for the Lower Basin diversions (MWD, SNWS, CAP) by using an alternative (surplus) demand schedules for these diversions.

Strategy 2: The Lake Mead elevation surplus determination is computed as follows

Step 1: determine Lake Mead's elevation at the beginning of the water year (October 1)

Step 2: if Lake Mead's elevation is greater than the trigger (input by the user), determine a surplus

Step 3: apply the following limits

• if the computed annual surplus release is zero or less, then there is no surplus determined

• if the computed annual surplus release is greater than zero, then the surplus flag is triggered.

Step 4: set new diversion schedules for the Lower Basin diversions (MWD, SNWS, CAP) by using an alternative (surplus) demand schedules for these diversions.

The user inputs the initial trigger elevation corresponding to a percentage of Lake Mead's allowed maximum capacity on October 1. Currently, this trigger elevation can be changed once during the run. For example, the user may input the initial trigger elevation to 60% of Lake Mead's maximum capacity, and then in some future year, reset this trigger to 80% of Lake Mead's maximum capacity.

Strategy 3: The Dave Steven's surplus strategy is determined as follows

Step 1: determine maximum contents for Flaming Gorge, Blue Mesa, Navajo, Powell and Mead.

Step 2: determine the current contents for Flaming Gorge, Blue Mesa, Navajo, Powell and Mead.

Step 3: determine variable SPACE by SPACE = maximum contents - current contents

Step 4: if SPACE/1000 is less than the space required (user input as 7.35 maf), determine a surplus. The 7.35 maf is the January 1 minimum vacant space required of 5.35 maf plus a 2 maf buffer.

Step 5: apply the following limits

• if the computed annual surplus release is zero or less, then there is no surplus determined

• if the computed annual surplus release is greater than zero, then the surplus flag is triggered.

Step 6: set new diversion schedules for the Lower Basin diversions (MWD, SNWS, CAP) by using an alternative (surplus) demand schedules for these diversions.

A.6. Lower Basin Shortage Strategy

The shortage determination strategy is based on comparing the January 1 Lake Mead elevation to a user input trigger elevation. The shortage is imposed on January 1 of each year a shortage is triggered and remains in effect for that calendar year.

Only CAP and SNWS have their diversion reduced below their normal schedule when a shortage is triggered. Currently CAP is assumed to divert 800,000 acre-feet if a shortage is triggered. SNWS is reduced by 4% of the CAP reduction if Nevada has reached full use of their 0.3 maf apportionment.

$$SNWS_{short} = SNWS_{norm} - 0.04*(CAP_{norm}-CAP_{short})$$

MWD and others do not take a shortage.

Shortages for CAP and SNWS continue until Mead is empty, at which time CAP delivery is equal to net inflow. If CAP delivery is reduced to zero, all entities within the United States and Mexico have shortages imposed. Shortages to Mexico consist of shorting Mexico proportionately to the shortages imposed on United States users.

$$MEX_{short} = Mex_{norm} * (U.S._{reductions}/U.S._{norm})$$

A.7. New Surplus Strategy

Step 1: determine Lake Mead's content or elevation at the beginning of the water year (October 1)

Step 2: if Lake Mead's elevation is greater than the trigger content or elevation (input by the user), a surplus is determined

It is assumed that the trigger elevation or content will be computed external to the model and input as data.

This strategy is based on the historical critical period of inflow to arrive at the determination of the required content or elevation. This strategy determined the minimum required elevation or content at Lake Mead to not allow Lake Mead's elevation to draw down below a specified elevation during the critical hydrology period.

Appendix B

Rules Written in *RSS's* Rule Language

B.1. Mead Flood Control

```
{-------------------------------------------------------------------------}

POLICY out1 TO_DETERMINE outflow FOR mead

BEGIN

IF now () <= JULY AND

   clear_cache ("trial_fc_rel") = 0.0 AND

   clear_cache ("release_level_index") = 0.0

THEN

   outflow = max (mead.outflow, flood_control_release)

END

{-------------------------------------------------------------------------}

POLICY fc_rel1 TO_DETERMINE flood_control_release FOR mead

BEGIN

IF now () <= JULY AND

   trial_fc_rel AND

   trial_fc_rel <= cfs_to_kaf (now (), now_year (),
                               nth (read_cache ("release_level_index"),
                                    &meadt.release_level)) AND

   read_cache ("release_level_index") > 1 AND

   trial_fc_rel > cfs_to_kaf (now (), now_year (),
                              nth (read_cache ("release_level_index") - 1,
                                   &meadt.release_level))

THEN

   flood_control_release = cache ("flood_control_release", trial_fc_rel)

END
```

```
{-------------------------------------------------------------------------}

POLICY fc_rel2 TO_DETERMINE flood_control_release FOR mead

BEGIN

IF now () <= JULY AND

   trial_fc_rel AND

   trial_fc_rel <= cfs_to_kaf (now (), now_year (),
                               nth (read_cache (“release_level_index”),
                                    &meadt.release_level)) AND

   read_cache (“release_level_index”) = 1

THEN

   flood_control_release = cache (“flood_control_release”,
                                  max (0.0, trial_fc_rel))

END

{-------------------------------------------------------------------------}

POLICY fc_rel3 TO_DETERMINE flood_control_release FOR mead

BEGIN

IF now () <= JULY AND

   trial_fc_rel AND

   trial_fc_rel <= cfs_to_kaf (now (), now_year (),
                               nth (read_cache (“release_level_index”),
                                    &meadt.release_level)) AND

   read_cache (“release_level_index”) > 1 AND

   trial_fc_rel <= cfs_to_kaf (now (), now_year (),
                               nth (read_cache (“release_level_index”) - 1,
                                    &meadt.release_level))

THEN

   flood_control_release =
      cache (“flood_control_release”,
             max (trial_fc_rel,
                  cfs_to_kaf (now (), now_year (),
                              nth (read_cache (“release_level_index”) - 1,
                                   &meadt.release_level))))

END

{-------------------------------------------------------------------------}

POLICY fc_rel4 TO_DETERMINE flood_control_release FOR mead

BEGIN
```

```
IF now () <= JULY AND

   trial_fc_rel AND

   trial_fc_rel > cfs_to_kaf (now (), now_year (),
                              nth (read_cache (“release_level_index”),
                                                &meadt.release_level))

THEN

   flood_control_release = reason (“flood_control_release”, &mead)

END

{-------------------------------------------------------------------------}
{  Mead Ending Storage                                                     }
{-------------------------------------------------------------------------}
POLICY end_storage TO_DETERMINE ending_storage FOR mead

BEGIN

IF now () >= AUGUST

THEN

   ending_storage =
      min (mead.live_capacity - meadt.min_flood_space,
           mead.live_capacity - ub.required_space + ub_creditable_space)

END

{-------------------------------------------------------------------------}

POLICY ub_cr_sp TO_DETERMINE ub_creditable_space FOR mead

BEGIN

   ub_creditable_space =
      min (ub.powell_creditable_space,
           (powell.live_capacity-powell.beginning_storage)) +

      min (ub.navajo_creditable_space,
           (navajo.live_capacity-navajo.beginning_storage)) +

      min (ub.blue_mesa_creditable_space,
           (blue_mesa.live_capacity - blue_mesa.beginning_storage)) +

      min (ub.flaming_gorge_creditable_space,
           (flaming_gorge.live_capacity - flaming_gorge.beginning_storage)) +

      min (ub.fontenelle_creditable_space,
           (fontenelle.live_capacity - fontenelle.beginning_storage)) +

      min (ub.mead_creditable_space,
           (mead.live_capacity-mead.beginning_storage))

END
```

```
{------------------------------------------------------------------------}

POLICY rli_1 TO_DETERMINE release_level_index FOR mead

BEGIN

IF check_cache (“release_level_index”) = 0.0

THEN

   release_level_index = cache (“release_level_index”, 2)

END

{------------------------------------------------------------------------}

POLICY rli_2 TO_DETERMINE release_level_index FOR mead

BEGIN

IF check_cache (“release_level_index”) <> 0.0

THEN

   release_level_index = cache (“release_level_index”,
                                read_cache (“release_level_index”) + 1)

END

{------------------------------------------------------------------------}

POLICY t_fc_rel TO_DETERMINE trial_fc_rel FOR mead

BEGIN

IF release_level_index

THEN

   trial_fc_rel =
      cache (“trial_fc_rel”,
             mead_max_forecast -
             available_space -
             combined_bstorage_change -
             reason (“average_evaporation”, &mead) -
             reason (“average_evaporation”, &powell) -
             nevada_depletion -
             (JULY - now ()) *
                cfs_to_kaf (now (), now_year (),
                            nth (read_cache (“release_level_index”),
                                 &meadt.release_level)))

END

{------------------------------------------------------------------------}

POLICY lake_mead_forecast TO_DETERMINE mead_max_forecast FOR mead

BEGIN
```

```
   mead_max_forecast = reason (“lake_powell_forecast”, &powell) -
                       nth (AUGUST, &powellt.average_natural_inflow) -
                       nth (SEPTEMBER, &powellt.average_natural_inflow) +
                       nth (now (), &meadt.forecast_constants) +
                       sum (&meadt.average_trib_natural_inflow, now (), JULY)

END

{-------------------------------------------------------------------------}

POLICY available_space TO_DETERMINE available_space FOR mead

BEGIN

   available_space = mead.live_capacity +
                     powell.live_capacity -
                     mead.beginning_storage -
                     powell.beginning_storage -
                     1500.

END

{-------------------------------------------------------------------------}

POLICY av_evap TO_DETERMINE average_evaporation FOR reservoir

BEGIN

   average_evaporation =
      (sum (&reservoir.evaporation_coefficients, now (), JULY) / 1000.) *
      area_from_contents (&reservoir,
                          (reservoir.live_capacity +
                           reservoir.beginning_storage) / 2.0)

END

{-------------------------------------------------------------------------}

POLICY nev_depletion TO_DETERMINE nevada_depletion FOR mead

BEGIN

   nevada_depletion = sum (&Md.diversion, now (), JULY) *
                      (1 - meadt.nevada_rflow_fraction)

END

{-------------------------------------------------------------------------}

POLICY comb_bs_ch TO_DETERMINE combined_bstorage_change FOR mead

BEGIN

   combined_bstorage_change =
      mead.bank_storage_coefficient *
         (mead.live_capacity - mead.beginning_storage - 1500.) +
      powell.bank_storage_coefficient *
         (powell.live_capacity - powell.beginning_storage)
```

```
END

{-------------------------------------------------------------------------}

POLICY powell_fcast1 TO_DETERMINE lake_powell_forecast FOR powell

BEGIN

IF now () <= JULY

THEN

   lake_powell_forecast = sum (&powellt.natural_inflow, now (), JULY) +
                          nth (AUGUST, &powellt.average_natural_inflow) +
                          nth (SEPTEMBER, &powellt.average_natural_inflow) -
                          ub.annual_depletion *
                             nth (now (), &ub.dist_depletion) +
                          forecast_error

END

{-------------------------------------------------------------------------}

POLICY powell_fore2 TO_DETERMINE lake_powell_forecast FOR powell

BEGIN

IF now () = AUGUST

THEN

   lake_powell_forecast = nth (AUGUST, &powellt.average_natural_inflow) +
                          nth (SEPTEMBER, &powellt.average_natural_inflow) -
                          ub.annual_depletion *
                             nth (now (), &ub.dist_depletion)

END

{-------------------------------------------------------------------------}

POLICY powell_for3 TO_DETERMINE lake_powell_forecast FOR powell

BEGIN

IF now () = SEPTEMBER

THEN

   lake_powell_forecast = nth (SEPTEMBER, &powellt.average_natural_inflow) -
                          ub.annual_depletion *
                             nth (now (), &ub.dist_depletion)

END

{-------------------------------------------------------------------------}

POLICY forecast_error1 TO_DETERMINE forecast_error FOR powell
```

```
BEGIN

IF read_cache (“release_level_index”) > 2

THEN

   forecast_error = read_cache (“forecast_error”)

END

{-------------------------------------------------------------------------}

POLICY forecast_error2 TO_DETERMINE forecast_error FOR powell

BEGIN

IF start_month () <= JULY AND

   start_month () = now () AND

   read_cache (“release_level_index”) = 2

THEN

   forecast_error = cache (“forecast_error”, 0.0)

END

{-------------------------------------------------------------------------}

POLICY forecast_error3 TO_DETERMINE forecast_error FOR powell

BEGIN

IF now () = JANUARY AND

   read_cache (“release_level_index”) = 2

THEN

   forecast_error =
      cache (“forecast_error”,
             nth (1 + 4 * (now () - 1), &powellt.forecast_error_coeff) *
                sum (&powellt.natural_inflow, now (), JULY) +
             nth (3 + 4 * (now () - 1), &powellt.forecast_error_coeff) +
             nth (4 + 4 * (now () - 1), &powellt.forecast_error_coeff) *
                random_number)

END

{-------------------------------------------------------------------------}

POLICY forecast_error4 TO_DETERMINE forecast_error FOR powell

BEGIN

IF now () > JANUARY AND

   now () <= JUNE AND
```

```
   start_month () <> now () AND

   read_cache (“release_level_index”) = 2 AND

   abs (0.5 * read_cache (“forecast_error”)) <=
      abs (nth (1 + 4 * (now () - 1), &powellt.forecast_error_coeff) *
               sum (&powellt.natural_inflow, now (), JULY) +
            nth (2 + 4 * (now () - 1), &powellt.forecast_error_coeff) *
               read_cache (“forecast_error”) +
            nth (3 + 4 * (now () - 1), &powellt.forecast_error_coeff) +
            nth (4 + 4 * (now () - 1), &powellt.forecast_error_coeff) *
               random_number)

THEN

   forecast_error = cache(“forecast_error”, 0.5 * read_cache
(“forecast_error”))

END

{-------------------------------------------------------------------------}

POLICY forecast_error5 TO_DETERMINE forecast_error FOR powell

BEGIN

IF now () > JANUARY AND

   now () <= JUNE AND

   start_month () <> now () AND

   read_cache (“release_level_index”) = 2 AND

   abs (0.5 * read_cache (“forecast_error”)) >
      abs (nth (1 + 4 * (now () - 1), &powellt.forecast_error_coeff) *
               sum (&powellt.natural_inflow, now (), JULY) +
            nth (2 + 4 * (now () - 1), &powellt.forecast_error_coeff) *
               read_cache (“forecast_error”) +
            nth (3 + 4 * (now () - 1), &powellt.forecast_error_coeff) +
            nth (4 + 4 * (now () - 1), &powellt.forecast_error_coeff) *
               random_number)

THEN

   forecast_error =
      cache (“forecast_error”,
             nth (1 + 4 * (now () - 1), &powellt.forecast_error_coeff) *
                sum (&powellt.natural_inflow, now (), JULY) +
             nth (2 + 4 * (now () - 1), &powellt.forecast_error_coeff) *
                read_cache (“forecast_error”) +
             nth (3 + 4 * (now () - 1), &powellt.forecast_error_coeff) +
             nth (4 + 4 * (now () - 1), &powellt.forecast_error_coeff) *
                random_number)

END

{-------------------------------------------------------------------------}
```

```
POLICY forecast_error6 TO_DETERMINE forecast_error FOR powell

BEGIN

IF now () = JULY AND

   start_month () <> now () AND

   read_cache (“release_level_index”) = 2


THEN

   forecast_error= cache(“forecast_error”, 0.25 * read_cache
(“forecast_error”))

END

{-------------------------------------------------------------------------}

POLICY random_number1 TO_DETERMINE random_number FOR powell

BEGIN

IF read_cache (“release_level_index”) = 2

THEN

   random_number = cache (“random_number”, random ())

END

{-------------------------------------------------------------------------}

POLICY random_number2 TO_DETERMINE random_number FOR powell

BEGIN

IF read_cache (“release_level_index”) > 2

THEN

   random_number = read_cache (“random_number”)

END

{-------------------------------------------------------------------------}

POLICY ub_cont_live_cap TO_DETERMINE ratio FOR reservoir

BEGIN

   ratio = (fontenelle.beginning_storage +
            flaming_gorge.beginning_storage +
            blue_mesa.beginning_storage +
            morrow_point.beginning_storage +
            crystal.beginning_storage +
            navajo.beginning_storage +
```

```
            powell.beginning_storage) /
           (fontenelle.live_capacity +
            flaming_gorge.live_capacity +
            blue_mesa.live_capacity +
            morrow_point.live_capacity +
            crystal.live_capacity +
            navajo.live_capacity +
            powell.live_capacity)

END

{-------------------------------------------------------------------------}

POLICY fg_tc1 TO_DETERMINE ending_storage FOR flaming_gorge

BEGIN

IF powell.beginning_elevation <= 3510.

THEN

   ending_storage = contents_from_elev (&flaming_gorge, 5871.0)

END

{-------------------------------------------------------------------------}

POLICY fg_tc2 TO_DETERMINE ending_storage FOR flaming_gorge

BEGIN

IF ratio AND

   powell.beginning_elevation <= 3690. AND

   powell.beginning_elevation > 3510.

THEN

   ending_storage = max (flaming_gorge.live_capacity * ratio,
                         contents_from_elev (&flaming_gorge, 5946.))

END

{-------------------------------------------------------------------------}

POLICY bm_tc1 TO_DETERMINE ending_storage FOR blue_mesa

BEGIN

IF powell.beginning_elevation <= 3510.

THEN

   ending_storage = contents_from_elev (&blue_mesa, 7393.0)

END

{-------------------------------------------------------------------------}
```

```
POLICY bm_tc3 TO_DETERMINE ending_storage FOR blue_mesa

BEGIN

IF ratio AND

   powell.beginning_elevation > 3510. AND

   powell.beginning_elevation <= 3570.

THEN

   ending_storage= max (blue_mesa.live_capacity * ratio,
                        contents_from_elev (&blue_mesa, 7438.0))

END

{-------------------------------------------------------------------------}

POLICY nav_tc1 TO_DETERMINE ending_storage FOR navajo

BEGIN

IF powell.beginning_elevation <= 3510.

THEN

   ending_storage = contents_from_elev (&navajo, 5990.0)

END

{-------------------------------------------------------------------------}

POLICY nav_tc3 TO_DETERMINE ending_storage FOR navajo

BEGIN

IF ratio AND

   powell.beginning_elevation > 3510. AND

   powell.beginning_elevation <= 3570.

THEN

   ending_storage = max (navajo.live_capacity * ratio,
                         contents_from_elev (&navajo, 6010.0))

END

{-------------------------------------------------------------------------}

POLICY mp_tc1 TO_DETERMINE ending_storage FOR morrow_point

BEGIN

IF powell.beginning_elevation <= 3510.
```

```
THEN

   ending_storage = contents_from_elev (&morrow_point, 7100.0)

END

{-------------------------------------------------------------------------}

POLICY crys_tc1 TO_DETERMINE ending_storage FOR crystal

BEGIN

IF powell.beginning_elevation <= 3510.

THEN

   ending_storage = contents_from_elev (&crystal, 6700.0)

END

{-------------------------------------------------------------------------}

POLICY surplus1 TO_DETERMINE mead_annual_surplus FOR diversion

BEGIN

IF now () = OCTOBER AND

   surplus.surplus_probability > 0.0

THEN

   mead_annual_surplus =
      cache (“mead_annual_surplus”,
             max (0, (mead.beginning_storage + powell.beginning_storage -
                      mead.live_capacity - powell.live_capacity +
                      0.7 * 1.0725 * nth (SEPTEMBER, &ub.space_required)) +
                      value_from_prob (surplus.surplus_porbability,
                                       lees_ferry.mean_annual_nat_inflow,
                                       lees_ferry.std_annual_nat_inflow) -
                      reason (“ub_annual_depletion”, &powell) -
                      reason (“lb_annual_depletion”, &mead) -
                      ub.avg_annual_evaporation))

END

{-------------------------------------------------------------------------}

POLICY ub_depletion TO_DETERMINE ub_annual_depletion FOR node

BEGIN

   ub_annual_depletion = {annual_depletion +
                          reasonr (“annual_depletion”, &node, &node.inflow_a) +
                           reasonr (“annual_depletion”, &node, &node.inflow_b)}

END

{-------------------------------------------------------------------------}
```

```
POLICY lb_depletion TO_DETERMINE lb_annual_depletion FOR node

BEGIN

   lb_annual_depletion = {annual_depletion +
                          reasonr (“annual_depletion”, &node, &node.outflow) +
                         reasonr (“annual_depletion”, &node, &node.diversion)}

END

{-------------------------------------------------------------------------}

POLICY depletion TO_DETERMINE annual_depletion FOR nodes

BEGIN

   annual_depletion = sum (&node.diversion, JANUARY, DECEMBER) *
                      (1 - node.rflow_fraction)

END

{-------------------------------------------------------------------------}

POLICY surplus2 TO_DETERMINE mead_annual_surplus FOR diversion

BEGIN

IF now () <> OCTOBER AND

   surplus.surplus_probability > 0.0

THEN

   mead_annual_surplus = read_cache (“mead_annual_surplus”)

END

{-------------------------------------------------------------------------}

POLICY cap_surp_cap1 TO_DETERMINE surplus_capacity FOR cap

BEGIN

IF now_year () < 1996

THEN

   surplus_capacity = 0.0

END

{-------------------------------------------------------------------------}

POLICY cap_surp_cap2 TO_DETERMINE surplus_capacity FOR cap

BEGIN

IF now_year () = 1996
```

```
THEN
   surplus_capacity = 2468.0 {cfs}
END
{-------------------------------------------------------------------------}
POLICY cap_surp_cap3 TO_DETERMINE surplus_capacity FOR cap
BEGIN
IF now_year () = 1997
THEN
   surplus_capacity = 2624.0 {cfs}
END
{-------------------------------------------------------------------------}
POLICY cap_surp_cap4 TO_DETERMINE surplus_capacity FOR cap
BEGIN
IF now_year () = 1998
THEN
   surplus_capacity = 2763.0 {cfs}
END
{-------------------------------------------------------------------------}
POLICY cap_surp_cap5 TO_DETERMINE surplus_capacity FOR cap
BEGIN
IF now_year () = 1999 OR
   (now_year () > 1999 AND cap_surplus_flag = 0.0)
THEN
   surplus_capacity = 2901.0 {cfs}
END
{-------------------------------------------------------------------------}
POLICY cap_surp_cap6 TO_DETERMINE surplus_capacity FOR cap
BEGIN
IF now_year () > 1996 AND
```

```
   read_cache (“cap_surplus_flag”) = 1.0
THEN
   surplus_capacity = 2486.0 {cfs}
END
{-------------------------------------------------------------------------}
POLICY nevada_surp_cap1 TO_DETERMINE surplus_capacity FOR nevada
BEGIN
IF now_year () < 2005
THEN
   surplus_capacity = 0.0
END
{-------------------------------------------------------------------------}
POLICY nevada_surp_cap2 TO_DETERMINE nevada_surplus_capacity FOR nevada
BEGIN
IF now_year () >= 2005
THEN
   nevada_surplus_capacity = 700 {cfs}
END
{-------------------------------------------------------------------------}
POLICY mwd_surp_cap1 TO_DETERMINE surplus_capacity FOR mwd
BEGIN
   surplus_capacity = 1851 {cfs}
END
{-------------------------------------------------------------------------}
POLICY diversion1 TO_DETERMINE diversion FOR diversion
BEGIN
IF check_cache (“surplus_flag”) = 0.0
THEN
   diversion = diversion +
               surplus_flag *
               cfs_to_kaf (now (), now_year (),
```

```
                          reason (“surplus_capacity”, &diversion))

END

{-------------------------------------------------------------------------}

POLICY diversion2 TO_DETERMINE diversion FOR diversion

BEGIN

IF check_cache (“surplus_flag”) <> 0.0

THEN

   diversion = diversion +
               read_cache (“surplus_flag”) *
                           cfs_to_kaf (now (), now_year (),
                                    reason (“surplus_capacity”, &diversion))

END

{-------------------------------------------------------------------------}

{
POLICY mex_rel TO_DETERMINE outflow FOR havasu

BEGIN

   outflow = havasu.outflow +
             max (0.0, ((flood_control_release + mead_annual_surplus) *
                        havasu.outflow / mex_total) -
                       (reason (“surplus_capacity”, &cap) +
                        reason (“surplus_capacity”, &mwd) +
                        reason (“surplus_capacity”, &nevada) / 12.0) +
                        max (0.0, mex_annual_shortage *
                                  havasu.outflow /
                                  mex_total))

END
}

{-------------------------------------------------------------------------}

POLICY surplus_flag1 TO_DETERMINE surplus_flag FOR diversion

BEGIN

IF mead_annual_surplus > 0.0 OR

   flood_control_release > 0.0

THEN

  surplus_flag = cache (“surplus_flag”, 1.0)

END

{-------------------------------------------------------------------------}
```

```
POLICY surplus_flag2 TO_DETERMINE surplus_flag FOR diversion
BEGIN
IF now () = JANUARY AND
   mead_annual_surplus = 0.0 AND
   flood_control_release = 0.0
THEN
  surplus_flag = cache (“surplus_flag”, 0.0)
END
{-------------------------------------------------------------------------}
POLICY surplus_flag3 TO_DETERMINE surplus_flag FOR diversions
BEGIN
IF now () <> JANUARY AND
   read_cache (“mead_annual_surplus”) = 0.0 AND
   read_cache (“flood_control_release”) = 0.0
THEN
   surplus_flag = read_cache (“surplus_flag”)
END
{-------------------------------------------------------------------------}
POLICY cap_surplus_flag1 TO_DETERMINE surplus_flag FOR cap
BEGIN
IF now () = SEPTEMBER AND
   surplus_flag = 1.0
THEN
   cap_surplus_flag = 1.0
END
{-------------------------------------------------------------------------}
POLICY cap_surplus_flag2 TO_DETERMINE surplus_flag FOR cap
BEGIN
IF now () = SEPTEMBER AND
   surplus_flag = 0.0
```

```
THEN

   cap_surplus_flag = 0.0

END
```

B.2. Upper Basin Rule Curve

```
{-------------------------------------------------------------------------}
{  Flaming Gorge target contents                                          }
{-------------------------------------------------------------------------}

POLICY fg_tc_1 TO_DETERMINE target_contents FOR flaming_gorge

BEGIN

IF powell.evelation > 3690

THEN

   target_contents = live_capacity - target_space

END

{-------------------------------------------------------------------------}

POLICY fg_tc_2 TO_DETERMINE target_contents FOR flaming_gorge

BEGIN

IF powell.elevation <= 3690 AND
   powell.evelation > 3510

THEN

   target_contents = computed_target_contents

END

{-------------------------------------------------------------------------}

POLICY fg_tc_3 TO_DETERMINE target_contents FOR flaming_gorge

BEGIN

IF powell.elevation <= 3510

THEN

   target_contents = contents_from_elev (&flaming_gorge, 5871.0)

END

{-------------------------------------------------------------------------}
{  Blue Mesa target contents                                              }
{-------------------------------------------------------------------------}

POLICY bm_tc_1 TO_DETERMINE target_contents FOR blue_mesa

BEGIN

IF powell.evelation > 3570

THEN
```

```
   target_contents = live_capacity - target_space

END

{-------------------------------------------------------------------------}

POLICY bm_tc_2 TO_DETERMINE target_contents FOR blue_mesa

BEGIN

IF powell.evelation <= 3570 AND
   powell.evelation > 3510

THEN

   target_contents = computed_target_contents

END

{-------------------------------------------------------------------------}

POLICY bm_tc_3 TO_DETERMINE target_contents FOR blue_mesa

BEGIN

IF powell.evelation <= 3510

THEN

   target_contents = contents_from_elev (&blue_mesa, 7393.0)

END

{-------------------------------------------------------------------------}
{  Navajo target contents                                                 }
{-------------------------------------------------------------------------}

POLICY nv_tc_1 TO_DETERMINE target_contents FOR navajo

BEGIN

IF powell.evelation > 3570

THEN

   target_contents = live_capacity - target_space

END

{-------------------------------------------------------------------------}

POLICY nv_tc_2 TO_DETERMINE target_contents FOR navajo

BEGIN

IF powell.elevation <= 3570 AND
   powell.evelation > 3510

THEN
```

```
   target_contents = computed_target_contents

END

{-------------------------------------------------------------------------}

POLICY nv_tc_3 TO_DETERMINE target_contents FOR navajo

BEGIN

IF powell.elevation <= 3510

THEN

   target_contents = contents_from_elev (&navajo, 5990.0)

END

{-------------------------------------------------------------------------}
{  Morrow Point target contents                                           }
{-------------------------------------------------------------------------}

POLICY mp_tc_1 TO_DETERMINE target_contents FOR morrow_point

BEGIN

IF powell.evelation > 3510

THEN

   target_contents = live_capacity

END

{-------------------------------------------------------------------------}

POLICY mp_tc_2 TO_DETERMINE target_contents FOR morrow_point

BEGIN

IF powell.evelation <= 3510

THEN

   target_contents = contents_from_elev (&morrow_point, 7100.0)

END

{-------------------------------------------------------------------------}
{  Crystal target contents                                                }
{-------------------------------------------------------------------------}

POLICY cr_tc_1 TO_DETERMINE target_contents FOR crystal

BEGIN

IF powell.evelation > 3510
```

```
THEN

   target_contents = live_capacity

END

{-------------------------------------------------------------------------}

POLICY cr_tc_2 TO_DETERMINE target_contents FOR crystal

BEGIN

IF powell.evelation <= 3510

THEN

   target_contents = contents_from_elev (&crystal, 6700.0)

END

{-------------------------------------------------------------------------}
{  Target Space                                                           }
{-------------------------------------------------------------------------}

POLICY ts1 TO_DETERMINE target_space FOR reservoir

BEGIN

IF now () > JULY

THEN

   target_space = target_space_data

END

{-------------------------------------------------------------------------}

POLICY ts2 TO_DETERMINE target_space FOR reservoir

BEGIN

IF now () <= JULY

THEN

   target_space = live_capacity -
                  max (max_allowable_storage, live_capacity)

END

{-------------------------------------------------------------------------}

POLICY pri TO_DETERMINE predicted_remaining_inflow FOR reservoir

BEGIN

   predicted_remaining_inflow = average_weights *
                                remaining_virgin_average_flow
```

```
                              + actual_weights *
                                remaining_virgin_actual_flow

END

{-------------------------------------------------------------------------}

POLICY rmvavf TO_DETERMINE remaining_virgin_average_flow FOR reservoir

BEGIN

   remaining_virgin_average_flow = sum (&inflow, now (), JULY)

END

{-------------------------------------------------------------------------}

POLICY rmvacf TO_DETERMINE remaining_virgin_actual_flow FOR reservoir

BEGIN

   remaining_virgin_actual_flow = sum (&average_virgin_inflow, now (), JULY)

END

{-------------------------------------------------------------------------}
{  Target Contents                                                        }
{-------------------------------------------------------------------------}

POLICY tc_fg TO_DETERMINE computed_target_contents FOR flaming_gorge

BEGIN

   computed_target_contents = max (contents_from_elev (&flaming_gorge,
                                                        5871.0),
                                   (live_capacity * ratio))

END

{-------------------------------------------------------------------------}

POLICY tc_bm TO_DETERMINE computed_target_contents FOR blue_mesa

BEGIN

   computed_target_contents = max (contents_from_elev (&blue_mesa, 7393.0),
                                   (live_capacity * ratio))

END

{-------------------------------------------------------------------------}

POLICY tc_nv TO_DETERMINE computed_target_contents FOR navajo

BEGIN

   computed_target_contents = max (contents_from_elev (&navajo, 5990.0),
                                   (live_capacity * ratio))
```

```
END

{-------------------------------------------------------------------------}

POLICY ratio TO_DETERMINE ratio FOR reservoir

BEGIN

   ratio = (ub.contents - fontenelle.contents) /
           (ub.total_live_capacity - fontenelle.live_capacity)

END

{-------------------------------------------------------------------------}
{  Max Allowable Storage                                                   }
{-------------------------------------------------------------------------}

POLICY mas_fg TO_DETERMINE max_allowable_storage FOR flaming_gorge

BEGIN

   max_allowable_storage = contents -
                           release_needed +
                           inflow_at_green_river_below_fontenelle () +
                           exports_above_fontenelle () +
                           gains_on_the_green_river_above_glendale ()

END

{-------------------------------------------------------------------------}

POLICY mas_bm TO_DETERMINE max_allowable_storage FOR blue_mesa

BEGIN

   max_allowable_storage = contents -
                           release_needed +
                           inflow_at_taylor_river_above_taylor_park () +
                           gains_on_the_gunnison_river_above_blue_mesa ()

END

{-------------------------------------------------------------------------}

POLICY mas_nv TO_DETERMINE max_allowable_storage FOR navajo

BEGIN

   max_allowable_storage = contents -
                           release_needed +
                           flow_at_san_juan_river_near_archuleta_NM () +
                           demand_from_navajo ()

END

{-------------------------------------------------------------------------}

POLICY rn TO_DETERMINE release_needed FOR reservoir
```

```
BEGIN

   release_needed = max (minimum_release,
                         (contents -
                          live_capacity +
                          predicted_remaining_inflow) / (AUGUST - now ()))

END
```

Appendix C

Rules Written in Pseudo-Code

C.1. Mead Flood Control Rule

C.1.1. Main Rule

```
RULE_NAME: mead_flood_control
RULE_PRIORITY: 1
RULE_DEPENDENCIES: mead.elevation

OBJECT_VARIABLES

   FT______ mead.elevation[]
   AF/M____ mead.min_release[]

DECLARATIONS

   AF/M____ Flood_Control ()
   AF/M____ Runoff_Forecast_Season ()
   AF/M____ Space_Building ()

   FT______ min_flood_control_elevation = 1219.61

BEGIN

   IF mead.elevation[] >= min_flood_control_elevation THEN

      mead.min_release[] = Flood_Control ()

   ELSEIF CURRENT_MONTH <= JULY THEN

      mead.min_release[] = Runoff_Forecast_Season ()

   ELSEIF

      mead.min_release[] = Space_Building ()

   ENDIF

END
```

C.1.2. Functions

```
FUNCTION Flood_Control ()

OBJECT_VARIABLES

   FT______ mead.elevation[]
   AF/M____ mead.inflow[]
```

```
DECLARATIONS

   TABLE flood_control {
      {level  elevation  release}
      {units         FT      CFS}
      {1        1221.40    28000}
      {2        1226.90    40000}
      {3        1229.00    65000}
   }

BEGIN
   IF mead.elevation[] <= flood_control[1, elevation] THEN

      RETURN (MIN (mead.inflow[], flood_control[1, release]))

   ELSEIF mead.elevation[] <= flood_control[2, elevation] THEN

      RETURN (MIN (mead.inflow[], flood_control[2, release]))

   ELSEIF mead.elevation[] <= flood_control[3, elevation] THEN

      RETURN (MIN (mead.inflow[], flood_control[3, release]))

   ELSE

      RETURN (mead.inflow[])

   ENDIF

END


AF/M____ FUNCTION Runoff_Forecast_Season ()

OBJECT_VARIABLES

   AC-FT___ mead.max_storage[]
   AC-FT___ mead.storage[]
   AC______ mead.surface_area[]
   AC-FT___ powell.max_storage[]
   AC-FT___ powell.storage[]
   AC______ powell.surface_area[]
   FT/M____ powell.evaporation_coefficient[CURRENT_MONTH..JULY]
   FT/M____ mead.evaporation_coefficient[CURRENT_MONTH..JULY]
   DMNLESS_ mead.bank_storage_coefficient[]
   DMNLESS_ powell.bank_storage_coefficient[]

DECLARATIONS

   AC-FT___ Powell_Flood_Control_Evaporation (AC-FT___, AC-FT___,
                                              FT/M____)
   AC-FT___ Evaporation_Estimate (STRING, AC-FT___, AC-FT___, FT/M____)
   AC-FT___ Bank_Storage_Change (AC-FT___, AC-FT___, AF/M____)
   AC-FT___ SNWP_Consumptive_Use (DMNLESS_, DMNLESS_)
   AF/M____ Mead_Inflow_Forecast ()
   AF/M____ C_Volume_To_Flow (AC-FT___)
   AF/M____ CFS_TO_AF/M (CFS____)
```

```
   AC-FT___ current_available_space
   AF/M____ powell_evap
   AF/M____ mead_evap
   AC-FT___ mead_bank_storage
   AC-FT___ powell_bank_storage
   AC-FT___ SNWP_consumptive_use
   AC-FT___ flood_control_storage
   AF/M____ flood_control_release
   DMNLESS_ discharge_level
   AF/M____ assumed_release
   AF/M____ min_mead_release

   CFS_____ DATA release_rate {
      {0     0}
      {1 19000}
      {2 28000}
      {3 35000}
      {4 40000}
      {5 73000}
   }

BEGIN

   current_available_space = mead.max_storage[] -
                             mead.storage[] +
                             powell.max_storage[] -
                             powell.storage[]

   powell_evap = Evaporation_Estimate (“powell”,
                           powell.max_storage[],
                           powell.surface_area[],
                         powell.evaporation_coefficient[CURRENT_MONTH..JULY])

   mead_evap = Evaporation_Estimate (“mead”,
                             mead.max_storage[],
                             mead.surface_area[],
                           mead.evaporation_coefficient[CURRENT_MONTH..JULY])

   mead_bank_storage = Bank_Storage_Change (mead.max_storage[] -
                                            mead.storage[],
                                            flood_control_space[AUGUST],
                                            mead.bank_storage_coefficient)

   powell_bank_storage = Bank_Storage_Change (powell.max_storage[] -
                                              powell.storage[],
                                              0.0,
                                             powell.bank_storage_coefficient)

   SNWP_consumptive_use = SNWP_Consumptive_Use (CURRENT_MONTH, JULY)

   flood_control_storage = Mead_Inflow_Forecast () +
                           flood_control_space[AUGUST] -
                           current_available_space -
                           powell_evap -
                           mead_evap -
                           powell_bank_storage -
                           mead_bank_storage -
                           SNWP_consumptive_use
```

```
   flood_control_release = C_Volume_To_Flow (flood_control_storage)

   #
   # Try discharge levels (starting with lowest) until release
   # determined.
   #
   FOR discharge_level = 0 TO 4

      assumed_release = CFS_TO_AF/M (release_rate[discharge_level])

      min_mead_release = flood_control_release - assumed_release

      IF min_mead_release <= assumed_release THEN
         RETURN (CFS_TO_AF/M (release_rate[discharge_level + 1])
      ENDIF

   ENDFOR

   RETURN (min_mead_release)

END


AF/M____ FUNCTION Space_Building ()

OBJECT_VARIABLES

   AC-FT___ mead.max_storage[]
   AC-FT___ mead.storage[]
   AC-FT___ powell.max_storage[]
   AC-FT___ powell.storage[]
   AC-FT___ navajo.max_storage[]
   AC-FT___ navajo.storage[]
   AC-FT___ blue_mesa.max_storage[]
   AC-FT___ blue_mesa.storage[]
   AC-FT___ flaming_gorge.max_storage[]
   AC-FT___ flaming_gorge.storage[]
   AC-FT___ fontenelle.max_storage[]
   AC-FT___ fontenelle.storage[]

DECLARATIONS

   AF/M____ C_Volume_To_Flow (AC-FT___)
   AF/M____ CFS_TO_AF/M (CFS____)

   AC-FT___ mead_space
   AC-FT___ flood_control_space
   AC-FT___ powell_space
   AC-FT___ navajo_space
   AC-FT___ blue_mesa_space
   AC-FT___ fg_fo_space
   AC-FT___ ub_creditable_space
   AC-FT___ total_space
   AC-FT___ needed_mead_space
   AF/M____ min_mead_release

   CFS_____    minimum_release_from_mead = 28000

   AC-FT___ DATA max_creditable_space {
```

```
      {POWELL                        38500000}
      {NAVAJO                        10359000}
      {BLUE_MESA                     07485000}
      {FLAMING_GORGE_AND_FONTENELLE 15072000}
   }

BEGIN

   #
   # Mead alone has enough flood control space, set release to 0.0.
   #
   mead_space = mead.max_storage[] - mead.storage[]

   flood_control_space = flood_control_space[NEXT_MONTH]

   IF mead_space >= flood_control_space THEN
      RETURN (0.0)
   ENDIF

   #
   # Upper basin reservoirs and Mead have enough flood control space,
   # set release to 0.0.
   #
   powell_space = MIN (max_creditable_space[POWELL],
                       powell.max_storage[] - powell.storage[])

   navajo_space = MIN (max_creditable_space[NAVAJO],
                       navajo.max_storage[] - navajo.storage[])

   blue_mesa_space = MIN (max_creditable_space[BLUE_MESA],
                          blue_mesa.max_storage[] -
                          blue_mesa.storage[])

   fg_fo_space = MIN (max_creditable_space[FLAMING_GORGE_AND_FONTENELLE],
                      flaming_gorge.max_storage[] -
                      flaming_gorge.storage[] +
                      fontenelle.max_storage[] -
                      fontenelle.storage[])

   ub_creditable_space = powell_space +
                         navajo_space +
                         blue_mesa_space +
                         fg_fo_space

   total_space = mead_space + ub_creditable_space

   IF total_space >= flood_control_space THEN
      RETURN (0.0)
   ENDIF

   #
   # Upper basin reservoirs and Mead do not have enough flood control space,
   # release amount needed to reach minimum flood control space, up to 28000.
   #
   needed_mead_space = MAX (flood_control_space - total_space,
                            flood_control_space[AUGUST])

   min_mead_release = C_Volume_To_Flow (needed_mead_space - mead_space)
```

```
   RETURN (MIN (min_mead_release, CFS_TO_AF/M (minimum_release_from_mead)))

END


AC-FT___ FUNCTION Mead_Inflow_Forecast ()

OBJECT_VARIABLES

DECLARATIONS

   AC-FT___ Powell_Forecast (DIMENSIONLESS, DIMENSIONLESS)
   AC-FT___ Sum_Object_Values (STRING, CONSTANT, STRING, CONSTANT,
                               STRING, DIMENSIONLESS, DIMENSIONLESS)
   VOID     Report_Error (STRING)

   AC-FT___ mean_forecast
   AC-FT___ max_forecast

   AC-FT___ DATA forecast_constant {
      {JANUARY  4980000}
      {FEBRUARY 4260000}
      {MARCH    3600000}
      {APRIL    2970000}
      {MAY      2525000}
      {JUNE     2130000}
      {JULY     0750000}
   }

BEGIN

   IF CURRENT_MONTH <= JULY THEN
      mean_forecast = Powell_Forecast (CURRENT_MONTH, JULY) +
                      Sum_Object_Values (“intervening_inflow”,
                                         UPSTREAM_EXCLUDING,
                                         “hoover”,
                                         DOWNSTREAM_EXCLUDING,
                                         “lees_ferry”,
                                         CURRENT_MONTH, JULY)

      max_forecast = mean_forecast + forecast_constant[]

   ELSE

      Report_Error (“Mead_Inflow_Forecast called after July.”)

   ENDIF

   RETURN (max_forecast)

END
```

C.1.3. Data Shared within Rule

None applicable

C.2. Upper Basin Rule Curve Rule

C.2.1. Main Rule

```
RULE_NAME: powell_elevation
RULE_PRIORITY: 2
RULE_DEPENDENCIES: powell.elevation

OBJECT_VARIABLES

   _________ powell.elevation[]
   _________ powell.turbine_overload_elevation
   _________ powell.rated_power_head
   _________ powell.min_power_pool_elevation
   _________ flaming_gorge_data.target_storage[]
   _________ blue_mesa_data.target_storage[]
   _________ navajo_data.target_storage[]
   _________ morrow_point_data.target_storage[]
   _________ crystal_data.target_storage[]
   _________ flaming_gorge_data.max_storage[]
   _________ blue_mesa_data.max_storage[]
   _________ navajo_data.max_storage[]
   _________ morrow_point_data.max_storage[]
   _________ crystal_data.max_storage[]
   _________ flaming_gorge.rated_power_head_elev
   _________ blue_mesa.rated_power_head_elev
   _________ navajo.rated_power_head_elev
   _________ flaming_gorge.min_power_elevation
   _________ blue_mesa_data.min_power_elevation
   _________ navajo_data.min_power_elevation
   _________ morrow_point_data.min_power_elevation
   _________ crystal_data.min_power_elevation

DECLARATIONS

   _________ Target_Space (STRING)
   _________ C_Volume_At_Elevation (STRING, _________)
   _________ Upper_Basin_Storage ()
   _________ Upper_Basin_Max_Storage ()

   DIMENSIONLESS ratio

BEGIN

   IF powell.elevation[] > powell.turbine_overload_elevation THEN

      flaming_gorge_data.target_storage[] = flaming_gorge.max_storage[] -
                                            Target_Space (“flaming_gorge”)

      blue_mesa_data.target_storage[] = blue_mesa.max_storage[] -
                                        Target_Space (“blue_mesa”)

      navajo_data.target_storage[] = navajo.max_storage[] -
                                     Target_Space (“navajo”)

      morrow_point_data.target_storage[] = morrow_point.max_storage[]

      crystal_data.target_storage[] = crystal.max_storage[]
```

```
ELSEIF powell.elevation[] > powell.rated_power_head THEN

   ratio = Upper_Basin_Storage () / Upper_Basin_Max_Storage ()

   flaming_gorge_data.target_storage[] = flaming_gorge.max_storage[] -
                    MAX (C_Volume_At_Elevation (“flaming_gorge”,
                                        flaming_gorge.rated_power_head_elev),
                         flaming_gorge.max_storage[] * ratio)

   blue_mesa_data.target_storage[] = blue_mesa.max_storage[] -
                                      Target_Space (“blue_mesa”)

   navajo_data.target_storage[] = navajo.max_storage[] -
                                  Target_Space (“navajo”)

   morrow_point_data.target_storage[] = morrow_point.max_storage[]

   crystal_data.target_storage[] = crystal.max_storage[]

ELSEIF powell.elevation[] > powell.min_power_pool_elevation + 20 THEN

   ratio = Upper_Basin_Storage () / Upper_Basin_Max_Storage ()

   flaming_gorge_data.target_storage[] =
      MAX (C_Volume_At_Elevation (“flaming_gorge”,
                                  flaming_gorge.rated_power_head_elev),
            flaming_gorge.max_storage[] * ratio)

   blue_mesa_data.target_storage[] =
          MAX (C_Volume_At_Elevation (“blue_mesa”,
                                    blue_mesa.rated_power_head_elevation),
               blue_mesa.max_storage[] * ratio)

   navajo_data.target_storage[] =
             MAX (C_Volume_At_Elevation (“navajo”,
                                      navajo.rated_power_head_elevation),
                  navajo.max_storage[] * ratio)

   morrow_point_data.target_storage[] = morrow_point.max_storage[]

   crystal_data.target_storage[] = crystal.max_storage[]


ELSE

   flaming_gorge_data.target_storage[] =
      C_Volume_At_Elevation (“flaming_gorge”,
                             flaming_gorge.min_power_elevation)

   blue_mesa_data.target_storage[] = C_Volume_At_Elevation (“blue_mesa”,
                                           blue_mesa.min_power_elevation)

   navajo_data.target_storage[] = C_Volume_At_Elevation (“navajo”,
                                              navajo.min_power_elevation)

   morrow_point_data.target_storage[] =
```

```
            C_Volume_At_Elevation (“morrow_point”,
                                   morrow_point.min_power_elevation)

      crystal_data.target_storage[] = C_Volume_At_Elevation (“crystal”,
                                                  crystal.min_power_elevation)

   ENDIF

END
```

C.2.2. Functions

```
_________ FUNCTION Target_Space (STRING reservoir)

OBJECT_VARIABLES

   _________ flaming_gorge.inflow[CURRENT_MONTH..JULY]
   _________ blue_mesa.inflow[CURRENT_MONTH..JULY]
   _________ navajo.inflow[CURRENT_MONTH..JULY]
   _________ flaming_gorge.storage[]
   _________ blue_mesa.storage[]
   _________ navajo.storage[]
   _________ flaming_gorge.max_storage[]
   _________ blue_mesa.max_storage[]
   _________ navajo.max_storage[]
   _________ flaming_gorge.min_release[]
   _________ blue_mesa.min_release[]
   _________ navajo.min_release[]

DECLARATIONS

   _________ Upstream_Inflow (STRING reservoir)

   _________ remaining_average_virgin_inflow
   _________ remaining_actual_inflow
   _________ weighted_average
   _________ release_needed
   _________ expected_inflow
   _________ max_allowable_storage

BEGIN

   IF CURRENT_MONTH >= AUGUST THEN

      RETURN (target_space[reservoir, CURRENT_MONTH])

   ELSE

      remaining_average_virgin_inflow =
         average_virgin_inflow[reservoir, CURRENT_MONTH..JULY]

      remaining_actual_inflow = reservoir.inflow[CURRENT_MONTH..JULY]

      weighted_average = weights[virgin, CURRENT_MONTH] *
                         remaining_average_virgin_inflow +
                         weights[actual, CURRENT_MONTH] *
                         remaining_actual_inflow

      release_needed = (reservoir.storage[] -
```

```
                        reservoir.max_storage[] +
                        weighted_average) /
                       (AUGUST - CURRENT_MONTH)

      release_needed = MAX (release_needed, reservoir.min_release[])

      expected_inflow = Upstream_Inflow (reservoir)

      max_allowable_storage = reservoir.storage[] -
                              release_needed +
                              expected_inflow

      max_allowable_storage = MIN (max_allowable_storage,
                                   reservoir.max_storage[])

      RETURN (reservoir.max_storage[] - max_allowable_storage)

   ENDIF

END


_________ FUNCTION Upstream_Inflow (STRING reservoir)

OBJECT_VARIABLES

   _________ inflow[] UPSTREAM OF AND EXCLUDING flaming_gorge
   _________ intervening_inflow[] UPSTREAM OF AND INCLUDING
                                  flaming_gorge
   _________ inflow[] UPSTREAM OF AND INCLUDING blue_mesa
   _________ intervening_inflow[] UPSTREAM OF AND INCLUDING blue_mesa
   _________ inflow[] UPSTREAM OF AND INCLUDING navajo
   _________ intervening_inflow[] UPSTREAM OF AND INCLUDING navajo
   _________ navajo.diversion[]
   _________ navajo.percent_return_flow[]

DECLARATIONS

   _________ Sum_Object_Values (STRING, CONSTANT, STRING, CONSTANT,
                                STRING, DIMENSIONLESS, DIMENSIONLESS)
   VOID      Report_Error (STRING)

   _________ expected_inflow
   _________ expected_intervening_inflow
   _________ expected_consumptive_use

BEGIN

   IF reservoir == “flaming_gorge” THEN

      expected_inflow = Sum_Object_Values (“inflow”,
                                           UPSTREAM_EXCLUDING,
                                           “flaming_gorge”, NA, “”,
                                           CURRENT_MONTH,
                                           CURRENT_MONTH)
      expected_intervening_inflow =
                              Sum_Object_Values (“intervening_inflow”,
                                                 UPSTREAM_INCLUDING,
                                                 “flaming_gorge”,
```

```
                                                CURRENT_MONTH,
                                                CURRENT_MONTH)
      expected_consumptive_use = 0

   ELSEIF reservoir == “blue_mesa” THEN

      expected_inflow = Sum_Object_Values (“inflow”,
                                           UPSTREAM_INCLUDING,
                                           “blue_mesa”, NA, “”,
                                           CURRENT_MONTH,
                                           CURRENT_MONTH)
      expected_intervening_inflow = Sum_Object_Values (“intervening_inflow”,
                                                       UPSTREAM_INCLUDING,
                                                       “blue_mesa”,
                                                       CURRENT_MONTH,
                                                       CURRENT_MONTH)
      expected_consumptive_use = 0

   ELSEIF reservoir == “navajo” THEN

      expected_inflow = Sum_Object_Values (“inflow”,
                                           UPSTREAM_INCLUDING,
                                           “navajo”, NA, “”,
                                           CURRENT_MONTH,
                                           CURRENT_MONTH)
      expected_intervening_inflow = Sum_Object_Values (“intervening_inflow”,
                                                       UPSTREAM_INCLUDING,
                                                       “navajo”,
                                                       CURRENT_MONTH,
                                                       CURRENT_MONTH)
      expected_consumptive_use = navajo.diversion[] *
                                 (1.0 - navajo.percent_return_flow[])

   ELSE

      Report_Error (“Invalid reservoir specified in Upstream_Inflow.”)

   ENDIF

   RETURN (expected_inflow +
           expected_intervening_inflow +
           expected_consumptive_use)

END


_________ FUNCTION Upper_Basin_Storage ()

OBJECT_VARIABLES

   _________ flaming_gorge.storage[]
   _________ blue_mesa.storage[]
   _________ morrow_point.storage[]
   _________ crystal.storage[]
   _________ navajo.storage[]
   _________ powell.storage[]

DECLARATIONS
```

```
BEGIN

   RETURN (flaming_gorge.storage[] +
           blue_mesa.storage[] +
           morrow_point.storage[] +
           crystal.storage[] +
           navajo.storage[] +
           powell.storage[])

END


_________ FUNCTION Upper_Basin_Max_Storage ()

OBJECT_VARIABLES

   _________ flaming_gorge.max_storage[]
   _________ blue_mesa.max_storage[]
   _________ morrow_point.max_storage[]
   _________ crystal.max_storage[]
   _________ navajo.max_storage[]
   _________ powell.max_storage[]

DECLARATIONS

BEGIN

   RETURN (flaming_gorge.max_storage[] +
           blue_mesa.max_storage[] +
           morrow_point.max_storage[] +
           crystal.max_storage[] +
           navajo.max_storage[] +
           powell.max_storage[])

END
```

C.2.3. Data Shared within Rule

```
_________ DATA target_space {
   {reservoir     AUGUST SEPTEMBER OCTOBER NOVEMBER DECEMBER}
   {flaming_gorge    100       150     210      280      350}
   {blue_mesa         25        50     100      150      200}
   {navajo            20        50      80      115      160}
}



_________ DATA average_virgin_inflow {
   {reservoir     JANUARY  FEBRUARY  MARCH  APRIL    MAY   JUNE   JULY}
   {flaming_gorge    34.0      34.5   94.6  176.0  339.8  561.6  346.8}
   {blue_mesa        23.3      20.9   33.8   87.9  250.4  327.8  157.5}
   {navajo           18.8      24.6   69.3  176.9  297.3  284.7  120.1}
}



_________ DATA weights {
   {weight     JANUARY  FEBRUARY  MARCH  APRIL    MAY   JUNE   JULY}
   {actual         0.3       0.4    0.5    0.7    0.7    0.7    0.6}
```

```
   {virgin         0.7       0.6    0.5    0.3    0.3    0.3    0.4}
}
```

C.3. Lower Basin Surplus Rule

C.3.1. Main Rule

```
RULE_NAME: surplus_strategy_1
RULE_PRIORITY: 3 <greater than shortage_strategy>
RULE_DEPENDENCIES:

OBJECT_VARIABLES

DECLARATIONS

   _________      Determine_Surplus_Using_Strategy_1 ()
   _________      Current_Water_Year ()
   _________      Allocate_Surplus (_________)

   BOOLEAN        surplus_declared
   DIMENSIONLESS last_declared_surplus_year

BEGIN

   IF CURRENT_MONTH == OCTOBER THEN

      surplus_declared = Determine_Surplus_Using_Strategy_1 ()

      IF surplus_declared THEN

         last_declared_surplus_year = Current_Water_Year ()

      ENDIF

   ENDIF

   Allocate_Surplus (last_declared_surplus_year)

END


RULE_NAME: surplus_strategy_2
RULE_PRIORITY: 3 <greater than shortage_strategy>
RULE_DEPENDENCIES:

OBJECT_VARIABLES

DECLARATIONS

   _________      Determine_Surplus_Using_Strategy_2 ()
   _________      Current_Water_Year ()
   _________      Allocate_Surplus (_________)

   BOOLEAN        surplus_declared
   DIMENSIONLESS last_declared_surplus_year

BEGIN

   IF CURRENT_MONTH == OCTOBER THEN

      surplus_declared = Determine_Surplus_Using_Strategy_2 ()
```

```
      IF surplus_declared THEN

         last_declared_surplus_year = Current_Water_Year ()

      ENDIF

   ENDIF

   Allocate_Surplus (last_declared_surplus_year)

END


RULE_NAME: surplus_strategy_3
RULE_PRIORITY: 3 <greater than shortage_strategy>
RULE_DEPENDENCIES:

OBJECT_VARIABLES

DECLARATIONS

   _________      Determine_Surplus_Using_Strategy_3 ()
   _________      Current_Water_Year ()
   _________      Allocate_Surplus ()

   BOOLEAN       surplus_declared
   DIMENSIONLESS last_declared_surplus_year

BEGIN

   IF CURRENT_MONTH == OCTOBER THEN

      surplus_declared = Determine_Surplus_Using_Strategy_3 ()

      IF surplus_declared THEN

         last_declared_surplus_year = Current_Water_Year ()

      ENDIF

   ENDIF

   Allocate_Surplus (last_declared_surplus_year)

END
```

C.3.2. Functions

```
BOOLEAN FUNCTION Determine_Surplus_Strategy_1 ()

OBJECT_VARIABLES

   _________ powell.max_storage[]
   _________ mead.max_storage[]

DECLARATIONS

   _________ Lees_Ferry_Natural_Flow ()
```

```
   _________ Upper_Basin_Consumptive_Use ()
   _________ Lower_Basin_Consumptive_Use ()

   _________ total_storage
   _________ total_allowed_storage
   _________ surplus

   _________ powell_mead_space_percent = 0.70
   _________ bank_storage_coefficient = ?
   _________ surplus_limit = 200000

BEGIN

   total_storage = powell.storage[] + mead.storage[]

   total_allowed_storage = powell.max_storage[] +
                           mead.max_storage[] -
                           powell_mead_space_percent *
                           flood_control_space[]

   surplus = Lees_Ferry_Natural_Flow () -
             Upper_Basin_Consumptive_Use () -
             Lower_Basin_Consumptive_Use () -
             (total_allowed_storage - total_storage) *
             (1.0 + bank_storage_coefficient)

   IF surplus < surplus_limit THEN

      RETURN (FALSE)

   ELSE

      RETURN (TRUE)

   ENDIF

END


BOOLEAN FUNCTION Determine_Surplus_Using_Strategy_2 ()

OBJECT_VARIABLES

   _________ mead.max_storage[]
   _________ mead.elevation[]

DECLARATIONS

   _________ C_Volume_At_Elevation (STRING, _________)

   _________ mead_surplus_storage
   _________ mead_storage
   _________ surplus

   _________ mead_surplus_percent = ?

BEGIN

   mead_surplus_storage = (mead.max_storage[] -
```

```
                          flood_control_space[]) *
                          mead_surplus_percent

   mead_storage = C_Volume_At_Elevation (“mead”, mead.elevation[])

   surplus = mead_storage - mead_surplus_storage

   IF surplus <= 0 THEN

      RETURN (FALSE)

   ELSE

      RETURN (TRUE)

   ENDIF

END


BOOLEAN FUNCTION Determine_Surplus_Using_Strategy_3 ()

OBJECT_VARIABLES

   _________ flaming_gorge.max_storage[]
   _________ blue_mesa.max_storage[]
   _________ navajo.max_storage[]
   _________ powell.max_storage[]
   _________ mead.max_storage[]
   _________ flaming_gorge.storage[]
   _________ blue_mesa.storage[]
   _________ navajo.storage[]
   _________ powell.storage[]
   _________ mead.storage[]

DECLARATIONS

   _________ max_storage
   _________ storage
   _________ space_available
   _________ surplus

   _________ surplus_storage = ?

BEGIN

   max_storage = flaming_gorge.max_storage[] +
                 blue_mesa.max_storage[] +
                 navajo.max_storage[] +
                 powell.max_storage[] +
                 mead.max_storage[]

   storage = flaming_gorge.storage[] +
             blue_mesa.storage[] +
             navajo.storage[] +
             powell.storage[] +
             mead.storage[]

   space_available = (max_storage - storage) / 1000.0
```

```
   surplus = surplus_storage - space_available

   IF surplus <= 0 THEN

     RETURN (FALSE)

   ELSE

      RETURN (TRUE)

   ENDIF

END


_________ FUNCTION Lees_Ferry_Natural_Flow ()

OBJECT_VARIABLES

DECLARATIONS

   _________ p
   _________ t
   _________ x
   _________ flow

   _________ probability = ?
   _________ mean = ?
   _________ standard_deviation = ?
   _________ LIST c {? ? ?}
   _________ LIST d {? ? ?}

BEGIN

   IF probability < 0.0001 THEN

      RETURN (0.0)

   ELSEIF probability > 0.5 THEN

      p = 1.0 - probability

   ELSE

      p = probability

   ENDIF

   t = SQRT (-2.0 * ALOG (p))

   x = t - (c[0] + c[1]*t + c[2]*t^2) / (1 + d[0]*t + d[1]*t^2 + d[2]*t^3)

   IF probability < 0.5 THEN

      x = -x

   ENDIF
```

```
   flow = mean + x * standard_deviation

   RETURN (flow)

END


_________ FUNCTION Upper_Basin_Consumptive_Use ()

OBJECT_VARIABLES

DECLARATIONS

   _________ upper_basin_consumptive_use = ?

BEGIN

   RETURN (upper_basin_consumptive_use)

END


_________ FUNCTION Lower_Basin_Consumptive_Use ()

OBJECT_VARIABLES

   _________ normal_diversion[JANUARY..DECEMBER]
                              UPSTREAM OF AND EXCLUDING hoover AND
                              DOWNSTREAM OF AND INCLUDING mexico
   _________ percent_return_flow UPSTREAM OF AND EXCLUDING hoover AND
                              DOWNSTREAM OF AND INCLUDING mexico
   _________ SNWP_data.normal_diversion[JANUARY..DECEMBER]

DECLARATIONS

   _________ Sum_Object_Values (STRING, CONSTANT, STRING, CONSTANT,
                               STRING, DIMENSIONLESS, DIMENSIONLESS)
   AC-FT___ SNWP_Consumptive_Use (DIMENSIONLESS , DIMENSIONLESS )
   _________ Evaporation_For_Surplus (STRING)

   _________ hoover_release
   _________ southern_nevada_pumping
   _________ mohave_evaporation
   _________ havasu_evaporation

   _________ mead_evaporation = 900000
   _________ annual_average_glen_to_hoover_gains = 801000
   _________ annual_average_gains_and_losses_below_hoover = 427000

BEGIN

   hoover_release = Sum_Object_Values (“consumptive_use”,
                                       DOWNSTREAM_EXCLUDING, “hoover”,
                                       UPSTREAM_INCLUDING, “mexico”,
                                       JANUARY, DECEMBER)

   southern_nevada_pumping = SNWP_Consumptive_Use (JANUARY, DECEMBER)

   mohave_evaporation = Evaporation_For_Surplus (“mohave”)
```

```
   havasu_evaporation = Evaporation_For_Surplus (“havasu”)

   RETURN (hoover_release +
           southern_nevada_pumping +
           mead_evaporation -
           annual_average_glen_to_hoover_gains -
           annual_average_gains_and_losses_below_hoover +
           mohave_evaporation +
           havasu_evaporation)

END


_________ FUNCTION Evaporation_For_Surplus (STRING reservoir)

OBJECT_VARIABLES

   _________ mohave_data.target_storage[JANUARY..DECEMBER]
   _________ havasu_data.target_storage[JANUARY..DECEMBER]

DECLARATIONS

   _________ C_Elevation_At_Volume (STRING, _________)
   _________ C_Area_At_Elevation (STRING, _________)

   _________ elevation
   _________ area
   _________ evaporation

   _________ DATA evaporation_coefficient {
      {JANUARY   ?}
      {FEBRUARY  ?}
      {MARCH     ?}
      {APRIL     ?}
      {MAY       ?}
      {JUNE      ?}
      {JULY      ?}
      {AUGUST    ?}
      {SEPTEMBER ?}
      {OCTOBER   ?}
      {NOVEMBER  ?}
      {DECEMBER  ?}
   }

BEGIN

   evaporation = 0.0

   FOR month = JANUARY, DECEMBER

      elevation = C_Elevation_At_Volume (reservoir,
                                         reservoir_data.target_storage[month])
      area = C_Area_At_Elevation (reservoir, elevation)

      evaporation = evaporation +
                    area *
                    evaporation_coefficient[month]
```

```
   ENDFOR

   RETURN (evaporation)

END


VOID FUNCTION Allocate_Surplus (DIMENSIONLESS surplus_water_year)

OBJECT_VARIABLES

   _________ MWD_data.surplus_diversion[]
   _________ SNWP_data.surplus_diversion[]
   _________ CAP_data.surplus_diversion[]
   _________ MWD.diversion[]
   _________ SNWP.diversion[]
   _________ CAP.diversion[]

DECLARATIONS

   _________ In_Water_Year (DIMENSIONLESS)
   _________ In_Calendar_Year (DIMENSIONLESS)
   KAF       CFS_TO_KAF (CFS)

   DIMENSIONLESS SNWP_surplus_start_year = 2010
   DIMENSIONLESS CAP_surplus_start_year  = 2030

BEGIN

   IF In_Water_Year (surplus_water_year)   ||
      In_Calendar_Year (surplus_water_year)  THEN

      MWD.diversion[] = CFS_TO_KAF (MWD_data.surplus_diversion[])

      IF CURRENT_YEAR >= SNWP_surplus_start_year THEN

         SNWP.diversion[] = CFS_TO_KAF (SNWP_data.surplus_diversion[])

      ENDIF

   ENDIF

   IF In_Calendar_Year (surplus_water_year) &&
      CURRENT_YEAR >= CAP_surplus_start_year  THEN

      CAP.diversion[] = CAP_data.surplus_diversion[]

   ENDIF

END


BOOLEAN FUNCTION In_Water_Year (DIMENSIONLESS year)

OBJECT_VARIABLES

DECLARATIONS

BEGIN
```

```
   IF (CURRENT_MONTH >= OCTOBER   && CURRENT_YEAR == year - 1) ||
      (CURRENT_MONTH <= SEPTEMBER && CURRENT_YEAR == year)       THEN

      RETURN (TRUE)

   ELSE

      RETURN (FALSE)

   ENDIF

END


BOOLEAN In_Calendar_Year (DIMENSIONLESS year)

OBJECT_VARIABLES

DECLARATIONS

BEGIN

   RETURN (CURRENT_YEAR == year)

END


BOOLEAN FUNCTION Current_Water_Year ()

OBJECT_VARIABLES

DECLARATIONS

BEGIN

   IF (CURRENT_MONTH >= OCTOBER THEN

      RETURN (CURRENT_YEAR + 1)

   ELSE

      RETURN (CURRENT_YEAR)

   ENDIF

END
```

C.3.3. Data Shared within Rule

None applicable.

C.4. Reservoir Equalization Rule

C.4.1. Main Rule

```
RULE_NAME: Reservoir_Equalization
RULE_PRIORITY: 4
DEPENDENCIES:

OBJECT_VARIABLES

   _________ powell.storage[]
   _________ mead.storage[]
   _________ powell.max_storage[]
   _________ mead.max_storage[]
   _________ powell_data.min_objective_release[]
   _________ powell.release[JANUARY..CURRENT_MONTH]

DECLARATIONS

   _________ Compute_602a_Storage ()
   _________ Powell_Predicted_EOWY_Storage ()
   _________ Mead_Predicted_EOWY_Storage ()
   _________ Upper_Basin_Predicted_Storage (_________)
   _________ C_Compute_Power_Plant_Capacity (STRING)

   _________ 602a
   _________ powell_predicted_EOWY_storage
   _________ mead_predicted_EOWY_storage
   _________ upper_basin_predicted_storage
   BOOLEAN   equalization_required
   _________ additional_release
   _________ min_objective_release
   _________ powell_power_plant_capacity
   _________ amount_over_released
   _________ reduced_release

BEGIN

   #
   # Determine parameters for reservoir equalization.
   #
   602a = Compute_602a_Storage ()
   powell_predicted_EOWY_storage = Powell_Predicted_EOWY_Storage ()
   mead_predicted_EOWY_storage = Mead_Predicted_EOWY_Storage ()
   upper_basin_predicted_storage =
      Upper_Basin_Predicted_Storage (powell_predicted_EOWY_storage)

   #
   # Determine if reservoir equalization is necessary.
   #
   IF upper_basin_predicted_storage > 602a &&
      powell_predicted_EOWY_storage >
      mead_predicted_EOWY_storage THEN

      equalization_required = TRUE

   ELSE
```

```
   equalization_required = FALSE

ENDIF

#
# Spread additional releases from powell over water year.
#
IF equalization_required && CURRENT_MONTH <= SEPTEMBER THEN

   additional_release = ((powell.storage[] - mead.storage[]) * 0.5) /
                        (OCTOBER - CURRENT_MONTH)
   min_objective_release = min_objective_release + additional_release

ENDIF

#
# Reduce Powell's release, if upper basin storage will drop
# below 602a.
#
IF (upper_basin_predicted_storage - min_objective_release) <
    602a THEN

   min_objective_release = upper_basin_predicted_storage - 602a

ENDIF

#
# Reduce Powell's release, if Mead's storage will exceed its
# live capacity minus its exclusive flood control space.
#
IF (mead.storage[] + min_objective_release) >
    (mead.max_storage[] - flood_control_space[]) THEN

   min_objective_release = mead.max_storage[] -
                           mead.storage[] -
                           flood_control_space[]

ENDIF

#
# Release cannot exceed maximum release (normally power plant capacity).
#
powell_power_plant_capacity = C_Compute_Power_Plant_Capacity ("powell")
min_objective_release = MIN (min_objective_release,
                             powell_power_plant_capacity)


#
# Powell's predicted storage less than expected, reduce release.
#
IF equalization_required &&
   powell_predicted_EOWY_storage < mead_predicted_EOWY_storage THEN

   amount_over_released =
                  powell.release[JANUARY..CURRENT_MONTH] -
                powell_data.min_objective_release[JANUARY..CURRENT_MONTH]

   reduced_release = amount_over_released / (DECEMBER - NEXT_MONTH)
   min_objective_release = min_objective_release - reduced_release
```

```
   ENDIF

   powell_data.min_objective_release[] = min_objective_release

END
```

C.4.2. Functions

```
________ FUNCTION Compute_602a_Storage ()

OBJECT_VARIABLES

   _________ lees_ferry_data.natural_inflow[JANUARY..DECEMBER]
   _________ powell.min_power_pool[]

DECLARATIONS

   BOOLEAN       C_Beginning_Of_Run ()
   ________      Upper_Basin_Estimated_Depletion (DIMENSIONLESS,
                                                  DIMENSIONLESS)

   DIMENSIONLESS month_count
   _________     upper_basin_depletion
   _________     min_objective_release
   _________     natural_flow
   _________     602a

   DIMENSIONLESS percent_shortage = 6.12
   DIMENSIONLESS critical_period = 12

BEGIN

   #
   # Default, if run started in middle of year.
   #
   IF C_Beginning_Of_Run () && CURRENT_MONTH != JANUARY THEN

      RETURN (Upper_Basin_Storage ())

   ENDIF

   #
   # Compute this value once per year, otherwise just return it.
   #
   IF CURRENT_MONTH == JANUARY THEN

      month_count = critical_period * MONTHS_IN_YEAR

      upper_basin_depletion = Upper_Basin_Estimated_Depletion (1,
                                                               month_count)

      min_objective_release = min_objective_release[JANUARY..DECEMBER] *
                              critical_period

      natural_flow = lees_ferry_data.natural_inflow[JANUARY..DECEMBER] *
                     critical_period

      602a = upper_basin_depletion * (100.0 - percent_shortage) /
```

```
            100.0 +
            min_objective_release -
            natural_flow +
            powell.min_power_pool[]

   ENDIF

   RETURN (602a)

END


________ FUNCTION Predicted_Powell_EOWY_Storage ()

OBJECT_VARIABLES

   _________ powell.max_storage[]
   _________ powell.surface_area[]
   _________ powell.evaporation_coefficient[CURRENT_MONTH..SEPTEMBER]
   _________ powell.storage[]
   _________ powell.bank_storage_coefficient[]

DECLARATIONS

   _________ Evaporation_Estimate (STRING, _________, _________, DIMENSIONLESS)
   _________ Bank_Storage_Change (_________, _________, DIMENSIONLESS)
   _________ Powell_Inflow_Forecast ()

   _________ assumed_release
   _________ evaporation
   _________ bank_storage

BEGIN

   assumed_release = min_objective_release[CURRENT_MONTH..SEPTEMBER]

   evaporation = Evaporation_Estimate (“powell”,
                                       powell.max_storage[],
                                       powell.surface_area[])

powell.evaporation_coefficient[CURRENT_MONTH..SEPTEMBER])

   bank_storage = Bank_Storage_Change (powell.max_storage[] -
                                       powell.storage[],
                                       0.0,
                                       powell.bank_storage_coefficient[])

   RETURN (powell.storage[] +
           Powell_Inflow_Forecast () -
           assumed_release -
           evaporation -
           bank_storage)

END


________ FUNCTION Powell_Inflow_Forecast ()

OBJECT_VARIABLES
```

```
   _________ powell_data.natural_inflow[CURRENT_MONTH..JULY]

DECLARATIONS

   _________ Upper_Basin_Estimated_Depletion (DIMENSIONLESS, DIMENSIONLESS)
   _________ Forecast_Error ()

   _________ forecast

   _________ DATA average_natural_inflow {
      {JANUARY   ?}
      {FEBRUARY  ?}
      {MARCH     ?}
      {APRIL     ?}
      {MAY       ?}
      {JUNE      ?}
      {JULY      ?}
      {AUGUST    ?}
      {SEPTEMBER ?}
   }

BEGIN

   IF CURRENT_MONTH < AUGUST THEN

      forecast = powell_data.natural_inflow[CURRENT_MONTH..JULY] +
                 average_natural_inflow[AUGUST..SEPTEMBER] -
                Upper_Basin_Estimated_Depletion (CURRENT_MONTH, SEPTEMBER) +
                 Forecast_Error ()

   ELSEIF CURRENT_MONTH == AUGUST THEN

      forecast = average_natural_inflow[AUGUST..SEPTEMBER] -
                 Upper_Basin_Estimated_Depletion (AUGUST, SEPTEMBER) +
                 Forecast_Error ()

   ELSEIF CURRENT_MONTH == SPETEMBER THEN

      forecast = average_natural_inflow[SEPTEMBER] -
                 Upper_Basin_Estimated_Depletion (SEPTEMBER, SEPTEMBER) +
                 Forecast_Error ()

   ENDIF

   RETURN (forecast)

END


_________ FUNCTION Predicted_Mead_EOWY_Storage ()

OBJECT_VARIABLES

   _________ mead.max_storage[]
   _________ mead.surface_area[]
   _________ mead.evaporation_coefficient[CURRENT_MONTH..JULY]
   _________ mead.bank_storage_coefficient
   _________ mead.storage[]
```

```
   _________ powell_data.release[]
   _________ mead_data.release[]

DECLARATIONS

   _________ Evaporation_Estimate (STRING, _________, _________, DIMENSIONLESS)
   _________ Bank_Storage_Change (_________, _________, DIMENSIONLESS)

   _________ evaporation
   _________ bank_storage

BEGIN

   evaporation = Evaporation_Estimate (“mead”,
                             mead.max_storage[],
                             mead.surface_area[],
                           mead.evaporation_coefficient[CURRENT_MONTH..JULY])

   bank_storage = Bank_Storage_Change (mead.max_storage[] -
                                       mead.storage[],
                                       flood_control_space[AUGUST] *
                                       1000.0,
                                       mead.bank_storage_coefficient)

   RETURN (mead.storage[] +
           powell_data.release[] -
           mead_data.release[] -
           evaporation -
           bank_storage)

END


_________ FUNCTION Upper_Basin_Predicted_Storage
                                     (_________ powell_predicted_EOWY_storage)

OBJECT_VARIABLES

   _________ flaming_gorge.storage[]
   _________ blue_mesa.storage[]
   _________ navajo.storage[]

DECLARATIONS

BEGIN

   RETURN (powell_predicted_EOWY_storage +
           flaming_gorge.storage[] +
           blue_mesa.storage[] +
           navajo.storage[])

END


_________ FUNCTION Forecast_Error ()

OBJECT_VARIABLES

   _________ powell_data.natural_runoff[CURRENT_MONTH..JULY]
```

```
DECLARATIONS

   BOOLEAN  C_Beginning_Of_Run ()
   _________ Random_Mean_Deviation ()
   VOID     Report_Error (STRING)

   _________ error
   _________ previous_months_error

   _________ DATA forecast_error {
      {month    runoff_coef error_coef constant std_error_of_estimate}
      {JANUARY         0.70       0.00   -8.195                 1.270}
      {FEBRUARY        0.00       0.80   -0.278                 0.977}
      {MARCH           0.00       0.90    0.237                 0.794}
      {APRIL           0.00       0.76    0.027                 0.631}
      {MAY             0.00       0.85    0.132                 0.377}
      {JUNE}           0.24       0.79    0.150                 0.460}
   }

BEGIN

   IF CURRENT_MONTH == JANUARY || C_Beginning_Of_Run () THEN

      previous_months_error = 0.0

   ENDIF

   IF CURRENT_MONTH < JULY THEN

      error = forecast_error[CURRENT_MONTH, runoff_coef] *
              powell_data.natural_runoff[CURRENT_MONTH..JULY] +
              forecast_error[CURRENT_MONTH, error_coef] *
              previous_months_error +
              Random_Mean_Deviation () *
              forecast_error[CURRENT_MONTH, std_error_of_estimate] +
              forecast_error[CURRENT_MONTH, constant]

      IF CURRENT_MONTH == JUNE THEN

         error = MIN (error, previous_months_error * 0.5)

      ENDIF

   ELSEIF CURRENT_MONTH == JULY THEN

      error = previous_months_error * 0.25

   ELSEIF CURRENT_MONTH == AUGUST || CURRENT_MONTH == SEPTEMBER THEN

      error = 0.0

   ELSE

      Report_Error (“No forecast error for months October-December.”)

   ENDIF

   previous_months_error = error
```

```
   RETURN (error)

END


_________ FUNCTION Random_Mean_Deviation ()

OBJECT_VARIABLES

DECLARATIONS

BEGIN

END
```

C.4.3. Data Shared within Rule

```
_________ DATA min_objective_release {
   {JANUARY   ?}
   {FEBRUARY  ?}
   {MARCH     ?}
   {APRIL     ?}
   {MAY       ?}
   {JUNE      ?}
   {JULY      ?}
   {AUGUST    ?}
   {SEPTEMBER ?}
   {OCTOBER   ?}
   {NOVEMBER  ?}
   {DECEMBER  ?}
   {JANUARY   ?}
}
```

C.5. Lower Basin Shortage Rule

C.5.1. Main Rule

```
RULE_NAME: shortage_strategy
RULE_PRIORITY: 5 <less than surplus_strategy_n>
RULE_DEPENDENCIES:

OBJECT_VARIABLES

   _________ CAP_data.normal_diversion[JANUARY..DECEMBER]
   _________ SNWP_data.normal_diversion[JANUARY..DECEMBER]
   _________ MWD_data.normal_diversion[JANUARY..DECEMBER]
   _________ mexico_data.normal_diversion[JANUARY..DECEMBER]

DECLARATIONS

   _________ CAP_Level_1_Diversion ()
   _________ SNWP_Level_1_Diversion (_________, _________)
   _________ Additional_Shortages (_________, _________)
   _________ CAP_Level_2_Diversion (_________, _________)
   _________ SNWP_Level_2_Diversion (_________, _________)
   _________ Mexicos_Diversion (_________, _________)
   _________ MWDs_Diversion  (_________, _________, _________)

   _________ CAP_diversion
   _________ SNWP_diversion
   _________ MWD_diversion
   _________ mexico_diversion
   _________ additional_shortages

BEGIN

   IF CURRENT_MONTH == JANUARY THEN

      CAP_diversion    = CAP_data.normal_diversion[JANUARY..DECEMBER]
      SNWP_diversion   = SNWP_data.normal_diversion[JANUARY..DECEMBER]
      MWD_diversion    = MWD_data.normal_diversion[JANUARY..DECEMBER]
      mexico_diversion = mexico_data.normal_diversion[JANUARY..DECEMBER]

      IF mead.elevation[] <= level_1_trigger_elev THEN

         CAP_diversion  = CAP_Level_1_Diversion ()
         SNWP_diversion = SNWP_Level_1_Diversion (CAP_diversion,
                                                  SNWP_diversion)

         additional_shortages = Additional_Shortages (CAP_diversion,
                                                      SNWP_diversion)

         IF additional_shortage > 0 THEN

            CAP_diversion  = CAP_Level_2_Diversion (CAP_diversion,
                                                    additional_shortage)
            SNWP_diversion = SNWP_Level_2_Diversion (SNWP_diversion,
                                                     additional_shortage)

            IF CAP_diversion == 0 THEN
```

```
                mexico_diversion = Mexicos_Diversion (mexico_diversion)
                MWD_diversion = MWDs_Diversion (CAP_diversion,
                                                SNWP_diversion,
                                                MWD_diversion)

             ENDIF

          ENDIF

       ENDIF

   ENDIF

   Set_Diversion (“cap”, CAP_diversion)
   Set_Diversion (“snwp”, SNWP_diversion)
   Set_Diversion (“mwd”, MWD_diversion)
   Set_Diversion (“mexico”, mexico_diversion)

END


_________ FUNCTION CAP_Level_1_Diversion ()

OBJECT_VARIABLES

   _________ CAP_data.shortage_diversion[JANUARY..DECEMBER]

DECLARATIONS

BEGIN

   RETURN (CAP_data.shortage_diversion[JANUARY..DECEMBER])

END


_________ FUNCTION SNWP_Level_1_Diversion (_________ CAP_diversion,
                                           _________ SNWP_diversion)

OBJECT_VARIABLES

   _________ CAP_data.normal_diversion[JANUARY..DECEMBER]

DECLARATIONS

   _________ CAP_shortage
   _________ SNWP_shortage

   _________ level_1_SNWP_shortage_percent = 0.04
   _________ SNWP_consumptive_use = 285000

BEGIN

   CAP_shortage = CAP_data.normal_diversion[JANUARY..DECEMBER] -
                  CAP_diversion
   SNWP_shortage = level_1_SNWP_shortage_percent * CAP_shortage

   IF SNWP_diversion > SNWP_consumptive_use - SNWP_shortage THEN
```

```
      RETURN (SNWP_diversion - SNWP_shortage)

   ELSE

      RETURN (SNWP_diversion)

   ENDIF

END


_________ FUNCTION CAP_Level_2_Diversion (_________ CAP_diversion,
                                          _________ additional_shortage)

DECLARATIONS

   _________ diversion

   _________ level_2_SNWP_shortage_percent = 0.04

BEGIN

   diversion = CAP_diversion - (1.0 - level_2_SNWP_shortage_percent) *
                               additional_shortage

   RETURN (MAX (diversion, 0))

END


_________ FUNCTION SNWP_Level_2_Diversion (_________ SNWP_diversion,
                                           _________ additional_shortage)

DECLARATIONS

   _________ diversion

   _________ level_2_SNWP_shortage_percent = 0.04

BEGIN

   diversion = SNWP_diversion - level_2_SNWP_shortage_percent *
                                additional_shortage

   RETURN (MAX (diversion, 0))

END


_________ FUNCTION Additional_Shortages (_________ CAP_diversion,
                                         _________ SNWP_diversion)

DECLARATIONS

   _________ C_Elevation_At_Volume (STRING, _________)

   _________ projected_mead_storage
   _________ projected_mead_elevation
```

```
   _________ level_2_trigger_elev = 1050

BEGIN

   projected_mead_storage = mead.storage[] -
                            CAP_diversion -
                            SNWP_diversion
   projected_mead_elevation = C_Elevation_At_Volume (“mead”,
                                                     projected_mead_storage)

   IF projected_mead_elevation <= level_2_trigger_elev THEN

      RETURN (mead.storage[] - projected_mead_storage)

   ELSE

      RETURN (0.0)

   ENDIF

END


#
#  Alternate version
#
_________ FUNCTION Additional_Shortages (_________ CAP_diversion,
                                         _________ SNWP_diversion)

OBJECT_VARIABLES

   _________ CAP_data.normal_diversion[JANUARY..DECEMBER]
   _________ SNWP_data.normal_diversion[JANUARY..DECEMBER]

DECLARATIONS

   _________ C_Elevation_At_Volume (STRING, _________)

   _________ CAP_shortage
   _________ SNWP_shortage
   _________ projected_mead_storage
   _________ projected_mead_elevation

   _________ level_2_trigger_elev = 1050

BEGIN

   CAP_shortage = CAP_data.normal_diversion[JANUARY..DECEMBER] -
                  CAP_diversion
   SNWP_shortage = SNWP_data.normal_diversion[JANUARY..DECEMBER] -
                   SNWP_diversion

   projected_mead_storage = mead.storage[] +
                            CAP_shortage +
                            SNWP_shortage
   projected_mead_elevation = C_Elevation_At_Volume (“mead”,
                                                     projected_mead_storage)

   IF projected_mead_elevation <= level_2_trigger_elev THEN
```

```
      RETURN (mead.storage[] - projected_mead_storage)

   ELSE

      RETURN (0.0)

   ENDIF

END


________ FUNCTION Mexicos_Diversion (________ CAP_shortage,
                                     ________ SNWP_shortage,
                                     ________ mexico_diversion)

OBJECT_VARIABLES

   _________ CAP_data.normal_diversion[JANUARY..DECEMBER]
   _________ SNWP_data.normal_diversion[JANUARY..DECEMBER]

DECLARATIONS

   _________ CAP_shortage
   _________ SNWP_shortage
   _________ US_shortage_percent
   _________ diversion

BEGIN

   CAP_shortage = CAP_data.normal_diversion[JANUARY..DECEMBER] -
                  CAP_diversion
   SNWP_shortage = SNWP_data.normal_diversion[JANUARY..DECEMBER] -
                   SNWP_diversion

   US_shortage_percent = (Upper_Basin_Shortage () +
                          CAP_shortage + SNWP_shortage) /
                          (Upper_Basin_Uses () + Lower_Basin_Uses ())

   diversion = mexico_diversion * US_shortage_percent

   RETURN (diversion)

END


_________ FUNCTION MWD_Diversion (_________ CAP_diversion,
                                  _________ SNWP_diversion,
                                  _________ MWD_diversion)

OBJECT_VARIABLES

   _________ CAP_data.normal_diversion[JANUARY..DECEMBER]
   _________ SNWP_data.normal_diversion[JANUARY..DECEMBER]
   _________ mexico_data.normal_diversion[JANUARY..DECEMBER]
   _________ mexico.diversion[JANUARY..DECEMBER]

DECLARATIONS
```

```
   _________ CAP_shortage
   _________ SNWP_shortage
   _________ mexico_shortage
   _________ total_shortage
   _________ remaining_shortage

BEGIN

   CAP_shortage = CAP_data.normal_diversion[JANUARY..DECEMBER] -
                  CAP_diversion

   SNWP_shortage = SNWP_data.normal_diversion[JANUARY..DECEMBER] -
                   SNWP_diversion

   mexico_shortage = mexico_data.normal_diverion[JANUARY..DECEMBER] -
                     mexico.diversion[JANUARY..DECEMBER]

   total_shortage = SNWP_shortage + CAP_shortage + mexico_shortage

   remaining_shortage = total_shortage - additional_shortage

   IF remaining_shortage > 0 THEN

      RETURN (MAX (MWD_diversion - remaining_shortage, 0.0))

   ELSE

      RETURN (MWD_diversion)

   ENDIF

END


_________ FUNCTION Level_1_Shortage ()

OBJECT_VARIABLES

DECLARATIONS

   _________ level_1_trigger_elev = ?

BEGIN

   RETURN ()

END


_________ FUNCTION Level_2_Shortage ()

OBJECT_VARIABLES

   _________ mead.storage[]

DECLARATIONS

   _________ C_Elevation_At_Volume (STRING, _________)
```

```
   _________ projected_mead_storage
   _________ projected_mead_elevation
   _________ level_2_trigger_elev = ?

BEGIN

   #
   # Get information on Mead's storage.
   #
   projected_mead_storage = mead.storage[] +
                            CAP_shortage +
                            SNWP_shortage

   projected_mead_elevation = C_Elevation_At_Volume ("mead",
                                                     projected_mead_storage)

   RETURN (projected_mead_elevation <= level_2_trigger_elev)

END


_________ FUNCTION Level_3_Shortage (_________ CAP_shortage_diversion)

OBJECT_VARIABLES

DECLARATIONS

BEGIN

   RETURN (CAP_shortage_diversion == 0)

END


RULE_NAME: shortage_strategy
RULE_PRIORITY: 5 <less than surplus_strategy_n>
RULE_DEPENDENCIES:

OBJECT_VARIABLES

   _________ mead.elevation[]
   _________ CAP_data.normal_diversion[JANUARY..DECEMBER]
   _________ CAP_data.shortage_diversion[JANUARY..DECEMBER]
   _________ SNWP_data.normal_diversion[JANUARY..DECEMBER]
   _________ SNWP_data.shortage_diversion[JANUARY..DECEMBER]
   _________ mexico_data.normal_diversion[JANUARY..DECEMBER]
   _________ mexico.diversion[JANUARY..DECEMBER]

DECLARATIONS

   _________ Set_CAP_Diversion (_________)
   _________ Set_SNWP_Diversion (_________)
   _________ C_Elevation_At_Volume (STRING, _________)
   _________ Upper_Basin_Shortage ()
   _________ Upper_Basin_Uses ()
   _________ Lower_Basin_Uses ()
   _________ Set_Mexico_Shortage_Diversion (_________)
   _________ Set_MWD_Diversion (_________)
```

```
   _________ CAP_normal_diversion
   _________ CAP_shortage_diversion
   _________ CAP_shortage
   _________ SNWP_normal_diversion
   _________ SNWP_shortage
   _________ projected_mead_storage
   _________ projected_mead_elevation
   _________ additional_shortage
   _________ US_shortage_percent
   _________ mexico_shortage
   _________ total_shortage
   _________ remaining_shortage
   _________ MWD_shortage

   _________ level_1_trigger_elev = ?
   _________ level_1_SNWP_shortage_percent = 0.04
   _________ SNWP_consumptive_use = 285000
   _________ level_2_trigger_elev = ?
   _________ level_2_SNWP_shortage_percent = 0.04

BEGIN

   #
   # In a shortage state for entire year.
   #
   IF CURRENT_MONTH == JANUARY &&
      mead.elevation[] <= level_1_trigger_elev THEN

      #
      # Determine CAP's shortage and monthly diversion.
      #
      CAP_normal_diversion = CAP_data.normal_diversion[JANUARY..DECEMBER]
      CAP_shortage_diversion = CAP_data.shortage_diversion[JANUARY..DECEMBER]

      Set_CAP_Diversion (CAP_shortage_diversion)

      #
      # Determine SNWP's shortage and diversion.
      #
      SNWP_normal_diversion = SNWP_data.normal_diversion[JANUARY..DECEMBER]
      SNWP_shortage_diversion =
                              SNWP_data.shortage_diversion[JANUARY..DECEMBER]
      SNWP_shortage = level_1_SNWP_shortage_percent * CAP_shortage

      IF SNWP_normal_diversion > SNWP_consumptive_use - SNWP_shortage THEN

         Set_SNWP_Diversion (SNWP_shortage_diversion)

      ELSE

         Set_SNWP_Diversion (SNWP_normal_diversion)

      ENDIF

      #
      # Get information on Mead's storage.
      #
      projected_mead_storage = mead.storage[] +
                               CAP_shortage +
```

```
                          SNWP_shortage
projected_mead_elevation = C_Elevation_At_Volume (“mead”,
                                          projected_mead_storage)

#
# Additional shortages are needed, based on Mead’s storage.
#
IF projected_mead_elevation <= level_2_trigger_elev THEN

   additional_shortage = mead.storage[] - projected_mead_storage

   SNWP_shortage_diversion = SNWP_shortage_diversion -
                             level_2_SNWP_shortage_percent *
                             additional_shortage

   SNWP_shortage_diversion = MAX (SNWP_shortage_diversion, 0)

   Set_SNWP_Diversion (SNWP_shortage_diversion)

   #
   # CAP takes remainder of additional shortage. If its
   # diversion is zero as a result, need to short Mexico.
   #
   CAP_shortage_diversion = CAP_shortage_diversion -
                            (1.0 - level_2_SNWP_shortage_percent) *
                            additional_shortage

   CAP_shortage_diversion = MAX (CAP_shortage_diversion, 0)

   Set_CAP_Diversion (CAP_shortage_diversion)

   IF CAP_shortage_diversion == 0 THEN

      CAP_shortage = CAP_normal_diversion - CAP_shortage_diversion
      SNWP_shortage = SNWP_normal_diversion - SNWP_shortage_diversion
      US_shortage_percent = (Upper_Basin_Shortage () +
                            CAP_shortage + SNWP_shortage) /
                            (Upper_Basin_Uses () +
                             Lower_Basin_Uses ())

      Set_Mexico_Shortage_Diversion (US_shortage_percent)

      #
      # If shortages to SNWP, CAP and Mexico are not enough, short MWD.
      #
     mexico_shortage = mexico_data.normal_diverion[JANUARY..DECEMBER] -
                       mexico.diversion[JANUARY..DECEMBER]

      total_shortage = SNWP_shortage +
                       CAP_shortage +
                       mexico_shortage

      remaining_shortage = total_shortage - additional_shortage

      IF remaining_shortage > 0 THEN

         MWD_shortage = MWD_data.normal_diversion - remaining_shortage
         Set_MWD_Shortage_Diversion (MWD_shortage)
```

```
            ENDIF

         ENDIF

      ENDIF

   ENDIF

END


RULE_NAME: shortage_strategy
RULE_PRIORITY: 5 <less than surplus_strategy_n>
RULE_DEPENDENCIES:

DECLARATIONS

   _________ Set_CAP_Shortage_Diversion (_________)
   _________ Set_SNWP_Shortage_Diversion (_________)
   _________ Set_SNWP_Normal_Diversion ()
   _________ C_Elevation_At_Volume (STRING, _________)
   _________ Upper_Basin_Shortage ()
   _________ Upper_Basin_Uses ()
   _________ Lower_Basin_Uses ()
   _________ Set_Mexico_Shortage_Diversion (_________)
   _________ Set_MWD_Shortage_Diversion (_________)

   _________ CAP_normal_diversion
   _________ CAP_shortage_diversion
   _________ CAP_shortage
   _________ SNWP_normal_diversion
   _________ SNWP_shortage
   _________ projected_mead_storage
   _________ projected_mead_elevation
   _________ additional_shortage
   _________ US_shortage_percent
   _________ mexico_shortage
   _________ total_shortage
   _________ remaining_shortage

   _________ level_1_trigger_elev = ?
   _________ level_1_SNWP_shortage_percent = 0.04
   _________ SNWP_consumptive_use = 285000
   _________ level_2_trigger_elev = ?
   _________ level_2_SNWP_shortage_percent = 0.04

BEGIN

   #
   # In a shortage state for entire year.
   #
   IF CURRENT_MONTH == JANUARY &&
      mead.elevation[] <= level_1_trigger_elev THEN

      #
      # Determine CAP's shortage and monthly diversion.
      #
      CAP_normal_diversion = CAP_data.normal_diversion[JANUARY..DECEMBER]
      CAP_shortage_diversion = CAP_data.shortage_diversion[JANUARY..DECEMBER]
```

```
CAP_shortage = CAP_normal_diversion - CAP_shortage_diversion

Set_CAP_Shortage_Diversion (CAP_shortage)

#
# Determine SNWP's shortage and diversion.
#
SNWP_normal_diversion = SNWP_data.normal_diversion [JANUARY..DECEMBER]
SNWP_shortage = level_1_SNWP_shortage_percent * CAP_shortage

IF SNWP_normal_diversion > SNWP_consumptive_use - SNWP_shortage THEN

   Set_SNWP_Shortage_Diversion (SNWP_shortage)

ELSE

   Set_SNWP_Normal_Diversion ()

ENDIF

#
# Get information on Mead's storage.
#
projected_mead_storage = mead.storage[] +
                         CAP_shortage +
                         SNWP_shortage
projected_mead_elevation = C_Elevation_At_Volume ("mead",
                                                  projected_mead_storage)

#
# Additional shortages are needed, based on Mead's storage.
#
IF projected_mead_elevation <= level_2_trigger_elev THEN

   additional_shortage = mead.storage[] - projected_mead_storage

   SNWP_shortage = SNWP_shortage +
                   shortage_data.level_2_SNWP_shortage_percent[0] *
                   additional_shortage

   IF SNWP_shortage > SNWP_normal_diversion THEN
      SNWP_shortage = SNWP_normal_diversion
   ENDIF

   Set_SNWP_Shortage_Diversion (SNWP_shortage)

   #
   # CAP takes remainder of additional shortage. If its
   # diversion is zero as a result, need to short Mexico.
   #
   CAP_shortage = CAP_shortage +
                  (1.0 - level_2_SNWP_shortage_percent) *
                  additional_shortage

   IF CAP_shortage > CAP_normal_diversion THEN

      CAP_shortage = CAP_normal_diversion

   ENDIF
```

```
         Set_CAP_Shortage_Diversion (CAP_shortage)

         IF CAP_shortage == CAP_normal_diversion THEN

            US_shortage_percent = (Upper_Basin_Shortage () +
                                   CAP_shortage + SNWP_shortage) /
                                  (Upper_Basin_Uses () +
                                   Lower_Basin_Uses ())

            Set_Mexico_Shortage_Diversion (US_shortage_percent)

            #
            # If shortages to SNWP, CAP and Mexico are not enough, short MWD.
            #
           mexico_shortage = mexico_data.normal_diverion[JANUARY..DECEMBER] -
                             mexico.diversion[JANUARY..DECEMBER]

            total_shortage = SNWP_shortage +
                             CAP_shortage +
                             mexico_shortage

            remaining_shortage = total_shortage -
                                 additional_shortage

            IF remaining_shortage > 0 THEN

               Set_MWD_Shortage_Diversion (remaining_shortage)

            ENDIF

         ENDIF

      ENDIF

   ENDIF

END
```

C.5.2. Functions

```
VOID FUNCTION Set_CAP_Diversion (_________ CAP_annual_diversion)

OBJECT_VARIABLES

   _________ CAP.diversion[JANUARY..DECEMBER]
   _________ CAP_data.monthly_percent[JANUARY..DECEMBER]

DECLARATIONS

   DIMENSIONLESS month

BEGIN

   FOR month = JANUARY, DECEMBER

      CAP.diversion[month] = CAP_data.monthly_percent[month] *
                             CAP_annual_diversion
```

```
   ENDFOR

END


VOID FUNCTION Set_CAP_Shortage_Diversion (_________ CAP_annual_shortage)

OBJECT_VARIABLES

   _________ CAP.diversion[JANUARY..DECEMBER]
   _________ CAP_data.normal_diversion[JANUARY..DECEMBER]
   _________ CAP_data.monthly_percent[JANUARY..DECEMBER]

DECLARATIONS

   DIMENSIONLESS month

BEGIN

   FOR month = JANUARY, DECEMBER

      CAP.diversion[month] = CAP_data.normal_diversion[month] -
                             CAP_data.monthly_percent[month] *
                             CAP_annual_shortage

   ENDFOR

END


VOID FUNCTION Set_SNWP__Diversion (_________ SNWP_annual_diversion)

OBJECT_VARIABLES

   _________ SNWP.diversion[JANUARY..DECEMBER]
   _________ SNWP_data.monthly_percent[JANUARY..DECEMBER]

DECLARATIONS

   DIMENSIONLESS month

BEGIN

   FOR month = JANUARY, DECEMBER

      SNWP.diversion[month] = SNWP_data.monthly_percent[month] *
                              SNWP_annual_diversion

   ENDFOR

END


VOID FUNCTION Set_SNWP_Shortage_Diversion (_________ SNWP_annual_shortage)

OBJECT_VARIABLES

   _________ SNWP.diversion[JANUARY..DECEMBER]
```

```
   _________ SNWP_data.normal_diversion[JANUARY..DECEMBER]
   _________ SNWP_data.monthly_percent[JANUARY..DECEMBER]

DECLARATIONS

   DIMENSIONLESS month

BEGIN

   FOR month = JANUARY, DECEMBER

      SNWP.diversion[month] = SNWP_data.normal_diversion[month] -
                              SNWP_data.monthly_percent[month] *
                              SNWP_annual_shortage

   ENDFOR

END


VOID FUNCTION Set_SNWP_Normal_Diversion ()

OBJECT_VARIABLES

   _________ SNWP.diversion[JANUARY..DECEMBER]
   _________ SNWP_data.normal_diversion[JANUARY..DECEMBER]

DECLARATIONS

   DIMENSIONLESS month

BEGIN

   FOR month = JANUARY, DECEMBER

      SNWP.diversion[month] = SNWP_data.normal_diversion[month]

   ENDFOR

END


VOID FUNCTION Set_Mexico_Shortage_Diversion (_________ US_shortage_percent)

OBJECT_VARIABLES

   _________ mexico.diversion[JANUARY..DECEMBER]
   _________ mexico_data.normal_diversion[JANUARY..DECEMBER]

DECLARATIONS

   DIMENSIONLESS month

BEGIN

   FOR month = JANUARY, DECEMBER

      mexico.diversion[month] = mexico_data.normal_diversion[month] *
                                US_shortage_percent
```

```
   ENDFOR

END


VOID FUNCTION Set_MWD_Diversion (_________ MWD_annual_diversion)

OBJECT_VARIABLES

   _________ MWD.diversion[JANUARY..DECEMBER]
   _________ MWD_data.monthly_percent[JANUARY..DECEMBER]

DECLARATIONS

   DIMENSIONLESS month

BEGIN

   FOR month = JANUARY, DECEMBER

      MWD.diversion[month] = MWD_data.monthly_percent[month] *
                             MWD_annual_diversion

   ENDFOR

END


VOID FUNCTION Set_MWD_Shortage_Diversion (_________ MWD_annual_shortage)

OBJECT_VARIABLES

   _________ MWD.diversion[JANUARY..DECEMBER]
   _________ MWD_data.normal_diversion[JANUARY..DECEMBER]
   _________ MWD_data.monthly_percent[JANUARY..DECEMBER]

DECLARATIONS

   DIMENSIONLESS month

BEGIN

   FOR month = JANUARY, DECEMBER

      MWD.diversion[month] = MWD_data.normal_diversion[month] -
                             MWD_data.monthly_percent[month] *
                             MWD_annual_shortage

   ENDFOR

END
```

C.5.3. Data Shared within Rule

None applicable.

C.6. Utility Functions

```
________ FUNCTION Evaporation_Estimate (STRING   reservoir,
                                        AC-FT___ beginning_storage,
                                        ________ ending_area,
                                        FT/M____ coefficient)

OBJECT_VARIABLES

DECLARATIONS

   _________ C_Elevation_At_Volume (STRING, _________)
   M3        KAF_TO_M3 (KAF)

   _________ beginning_elevation
   _________ beginning_area
   _________ average_area

BEGIN

   beginning_elevation = C_Elevation_At_Volume (reservoir, beginning_storage)
   beginning_area = C_Area_At_Elevation (reservoir, beginning_elevation)
   average_area = (beginning_area + ending_area) / 2.0

   RETURN (KAF_TO_M3 (average_area * coefficient / 1000.0))

END


_________ FUNCTION Bank_Storage_Change (_________ beginning_space,
                                        _________ ending_space,
                                        _________ coefficient)

OBJECT_VARIABLES

DECLARATIONS

BEGIN

   RETURN (coefficient * (beginning_space - ending_space))

END


_________ FUNCTION Upper_Basin_Estimated_Depletion (DIMENSIONLESS start_month,
                                                    DIMENSIONLESS end_month)

OBJECT_VARIABLES

   _________ diversion[JANUARY..DECEMBER] UPSTREAM AND INCLUDING powell

DECLARATIONS

   AC-FT___ Get_Object_Values (STRING, CONSTANT, STRING, CONSTANT,
                               STRING, DIMENSIONLESS, DIMENSIONLESS)

BEGIN

   RETURN (Sum_Object_Values (“diversion”, UPSTREAM_INCLUDING,
```

```
                                “powell”, NA, “”, start_month,
                                end_month))

END


AC-FT___ FUNCTION Sum_Object_Values (STRING        slot,
                                     CONSTANT      start_option,
                                     STRING        start_object,
                                     CONSTANT      end_option,
                                     STRING        end_object,
                                     DIMENSIONLESS start_month,
                                     DIMENSIONLESS end_month)

OBJECT_VARIABLES

DECLARATIONS

   _________     Get_Object_Names (STRING, CONSTANT, STRING, CONSTANT, STRING)
   STRING_LIST   object_list
   DIMENSIONLESS value = 0.0
   STRING        object_name

BEGIN

   IF slot == “consumptive_use” THEN

      object_list = Get_Object_Names (“diversion”, start_option,
                                      start_object, end_option, end_object)

      FOR object_name IN object_list

         FOR month = start_month, end_month

            value = value +
                    (1.0 - object_name.percent_return_flow[month]) *
                    object_name.diversion[month]

         ENDFOR

      ENDFOR

   ELSE

      object_list = Get_Object_Names (slot, start_option,
                                      start_object, end_option, end_object)

      FOR object_name IN object_list

         FOR month = start_month, end_month

            value = value + object_name.slot[month]

         ENDFOR

      ENDFOR

   ENDIF
```

```
   RETURN (value)

END


AC-FT___ FUNCTION SNWP_Consumptive_Use (start_month, end_month)

OBJECT_VARIABLES

   AF/M____ SNWP_data.normal_diversion

DECLARATIONS

   AC-FT___ SNWP_consumptive_use
   DMNLESS_ SNWP_return_flow = 0.38

BEGIN

   SNWP_consumptive_use = 0.0

   FOR month = start_month, end_month

      SNWP_consumptive_use = SNWP_consumptive_use +
                             (1 - SNWP_return_flow) *
                             SNWP_data.normal_diversion[month]

   ENDIF

   RETURN (SNWP_consumptive_use)

END
```

C.7. Data Shared by Multiple Rules

```
________ DATA flood_control_space {
   {JANUARY      ?}
   {FEBRUARY     ?}
   {MARCH        ?}
   {APRIL        ?}
   {MAY          ?}
   {JUNE         ?}
   {JULY         ?}
   {AUGUST    1.50}
   {SEPTEMBER 2.27}
   {OCTOBER   3.04}
   {NOVEMBER  3.81}
   {DECEMBER  4.58}
   {JANUARY   5.35}
}
```

C.8. Rule Syntax Conventions

The following table contains the syntactic conventions used in the pseudo-code.

String	Meaning	Example
ALL_CAPITALS	Reserved word in pseudo-code language	BEGIN
Only_First_Letters_Of_ Words_Capitalized (...)	Function written in pseudo-code	Space_Building ()
no_capitals	Local variable in pseudo-code	mead_evap
C_Only_First_Letters_Of_ Words_Capitalized (...)	Function written in C	C_Volume_To_Flow (flood_control_storage)
object.slot[]	Get slot value from object for current_month	mead.elevation[]
object.slot[MONTH]	Get slot value from object for MONTH	mead.elevation[JULY]
object.slot[MONTH1.. MONTH2]	Sum slot values from object from MONTH1 to MONTH2	mead.elevation[JULY..DECEMBER]
name[MONTH]	Get data from internal data structure	flood_control_space[AUGUST]
name[MONTH1..MONTH2]	Sum data from internal data structure from MONTH1 to MONTH2	flood_control_space [AUGUST..DECEMBER]

Appendix D

Potential Language Comparisons

D.1. CRSS Version of Upper Basin Rule Curve Rule

The following is a modified version of the CRSS rule. Its purpose is to codify the logic found in the previous example, not to suggest a particular syntax.

```
#
# Rationale: If Powell's elevation is above turbine overload, there is
# no power advantage in storing more water there, so set other upper
# basin reservoirs as high as possible.
#
# "Turbine overload elevation" = 3690 in code.
# "Max power head" = 3698 in Appendix A.
#
# Are the target_spaces for flood control? - Yes.
#
```

IF $\text{elevation}_{PO,\,mo} > \text{turbine\_overload\_elevation}_{PO}$ THEN

$\text{target\_storage}_{FG,\,mo}$ = $\text{live\_capacity}_{FG}$ - $\text{target\_space}_{FG,\,mo}$
$\text{target\_storage}_{BM,\,mo}$ = $\text{live\_capacity}_{BM}$ - $\text{target\_space}_{BM,\,mo}$
$\text{target\_storage}_{NAV,\,mo}$ = $\text{live\_capacity}_{NAV}$ - $\text{target\_space}_{NAV,\,mo}$
$\text{target\_storage}_{MP,\,mo}$ = $\text{live\_capacity}_{MP}$
$\text{target\_storage}_{CR,\,mo}$ = $\text{live\_capacity}_{CR}$

```
#
# Rationale: Powell's elevation is above rated head elevation, so try
# to keep other upper basin reservoirs at their rated head or above.
#
# Rated power head elevation = 3570.
#
```

ELSE IF $\text{elevation}_{PO,\,mo} > \text{rated\_power\_head}_{PO}$ THEN

$$\text{ratio} = \frac{\text{current\_contents}_{UB}}{\text{total\_capacity}_{UB}}$$

where UB = {FG, BM, MP, CR, NAV, POW}, but not FO

Table 3: Total UB live_capacity.

Reservoir	live_capacity
FO	344,800
FG	3,749,200

Table 3: Total UB live_capacity.

Reservoir	live_capacity
BM	829,500
MP	117,025
CR	17,536
NAV	1,696,000
POW	24,322,344
Total with FO	31,076,405
Total without FO	30,731,605

```
# Rated power head elevation at FG = 5946
# Note: volume at this elevation will need to be adjusted for
# sediment as the simulation runs.
target_storage_FG,mo = live_capacity_FG -
                    MAX (volume_at_FG (rated_power_head_elevation_FG),
                         live_capacity_FG * ratio)
target_storage_BM,mo = live_capacity_BM - target_space_BM,mo
target_storage_NAV,mo = live_capacity_NAV - target_space_NAV,mo
target_storage_MP,mo = live_capacity_MP
target_storage_CR,mo = live_capacity_CR
#
# Rationale: Powell's elevation is in "normal" operating range,
# therefore keep FG, BM and NAV in a range defined by how full the
# UB is.
#
# Note: Powell's minimum power pool elevation is 3490.
# Is 3510 used to "protect" 3490?
#
# Use ratio from previous case.
#
ELSE IF elevation_PO,mo > minimum_power_pool_elevation_PO + 20 THEN

# FG's rated power head = 5946.
# BM's rated power head = 7438.
# Assume NAV's rated power head = 6010.

target_storage_FG,mo = MAX (volume_at_FG (rated_power_head_elevation_FG),
                         live_capacity_FG * ratio)
target_storage_BM,mo = MAX (volume_at_BM (rated_power_head_elevation_BM),
                         live_capacity_BM * ratio)
target_storage_NAV,mo = MAX (volume_at_NAV (rated_power_head_elevation_NAV),
                         live_capacity_NAV * ratio)
target_storage_MP,mo = live_capacity_MP
target_storage_CR,mo = live_capacity_CR
#
# Rationale: Powell is approaching minimum power elevation (3490),
# so bring other UB reservoirs down to their minimum power
```

```
   # elevation to try to keep Glen on line.
   #
ELSE

   #
   # FG's minimum power elevation = 5871
   # BM's minimum power elevation = 5871
   # Assume NAV's minimum power elevation = 5871
   # MP's minimum power elevation = 5871
   # CR's minimum power elevation = 5871
   #
   target_storage_FG,mo = volume_at_FG (minimum_power_elevation_FG)
   target_storage_BM,mo = volume_at_BM (minimum_power_elevation_BM)
   target_storage_NAV,mo = volume_at_NAV (minimum_power_elevation_NAV)
   target_storage_MP,mo = volume_at_MP (minimum_power_elevation_MP)
   target_storage_CR,mo = volume_at_CR (minimum_power_elevation_CR)

ENDIF


#
# Determination of target_space applies only to BM, FG and NAV.
#
Determine target_space

IF august ≤ current_month ≤ december THEN

   use the following table:
```

Table 4: Target Space Data kac-ft.

Reservoir	Aug	Sep	Oct	Nov	Dec
BM	25	50	100	150	200
FG	100	150	210	280	350
NAV	20	50	80	115	160

```
   The location of this table may also be on a simulation object or in the
   rulebase.

ELSE

   1. Compute the remaining "average" virgin inflow into the
      reservoir from current month through July using the following
      equation and table:
```

$$I_{v,\,mo} = \sum_{mo\,=\,current\_month}^{july} \text{average virgin inflow}_{mo}$$

Table 5: "average" virgin inflow kac-ft.

Reservoir	Jan	Feb	Mar	Apr	May	Jun	Jul
BM	23.3	20.9	33.8	87.9	250.4	327.8	157.5
FG	34.0	34.5	94.6	176.0	339.8	561.6	346.8
NAV	18.8	24.6	69.3	176.9	297.3	284.7	120.1

The values in the above table should be computed once at the beginning of a run, using forcing hydrology in one or more tables which specify inflow for BM, FG and NAV. The location of this table may also be on a simulation object or in the rulebase.

2. Compute the remaining "actual" inflow into the reservoir from the current month through July using the following equation and data table in simulation:

$$I_{a,mo} = \sum_{mo = current\_month}^{july} \text{actual inflow}_{mo}$$

3. Use weighted average of the "actual" remaining inflow (input) and the "average" virgin remaining inflow to predict the remaining inflow from current month through July using the following equation and table:

$$I_{pred,mo} = C_v \cdot I_{v,mo} + C_a \cdot I_{a,mo}$$

Table 6: actual and virgin weights

Month	Actual (C_a)	Virgin (C_v)
Jan	0.3	0.7
Feb	0.4	0.6
Mar	0.5	0.5
Apr	0.7	0.3
May	0.7	0.3
Jun	0.7	0.3
Jul	0.6	0.4

The location of this table may also be on a simulation object or in the rulebase.

4. Compute the current month release needed to store this predicted inflow:

$$\text{release\_needed}_{mo} = \frac{\text{contents}_{mo} - \text{live\_capacity}_{mo} + I_{pred,mo}}{\text{august} - current\_month}$$

```
#
# Not sure if minimum_release varies per month.
#
IF release_needed_mo < minimum_release_<RES> THEN
   release_needed_mo = minimum_release_<RES>

5. Compute the maximum allowable storage in each reservoir that can
   still contain the predicted inflow.

# BM:
maximum_allowable_storage_mo = storage_mo -
                               release_needed_mo +
                               inflow_Taylor River above Taylor Park,mo +
                               gain_Gunnison River above Blue Mesa, mo

# FG:
maximum_allowable_storage_mo = storage_mo -
                               release_needed_mo +
                               inflow_Green River below Fontenelle, mo +
                               exports_above_Fontenelle, mo +
                               gains_Green River above Greendale, Ut, mo

# NAV:
maximum_allowable_storage_mo = storage_mo -
                               release_needed_mo +
                               flow_San Juan River near Archuleta, NM, mo +
                               demand_Navajo Reservoir, mo

IF maximum_allowable_storage_mo > live_capacity_<RES> THEN
   maximum_allowable_storage_mo = live_capacity_<RES>

6. Compute the target space (“drawdown”)
target_space_mo = live_capacity_mo - maximum_allowable_storage_mo
```

D.2. Simple Lex/Yacc Version of Upper Basin Rule Curve Rule

```
RULE_NAME: powell_elevation
RULE_PRIORITY: 1
RULE_DEPENDENCIES: powell.elevation

BEGIN

   IF powell.elevation > powell.turbine_overload_elevation THEN

      flaming_gorge_data.target_storage = flaming_gorge.live_capacity -
                                          Flaming_Gorge_Target_Space ();
      blue_mesa_data.target_storage = blue_mesa.live_capacity -
                                      Blue_Mesa_Target_Space ();
      navajo_data.target_storage = navajo.live_capacity -
                                   Navajo_Target_Space ();
      morrow_point_data.target_storage = morrow_point.live_capacity;
      crystal_data.target_storage = crystal.live_capacity;

   ELSE IF powell.elevation > powell.rated_power_head THEN

      ratio = Current_Upper_Basin_Storage () / Upper_Basin_Total_Capacity ();

      flaming_gorge_data.target_storage = flaming_gorge.live_capacity -
               MAX (C_Volume_At (“flaming_gorge.rated_power_head_elevation”),
                     flaming_gorge.live_capacity * ratio);
      blue_mesa_data.target_storage = blue_mesa.live_capacity -
                                      Blue_Mesa_Target_Space ();
      navajo_data.target_storage = navajo.live_capacity -
                                   Navajo_Target_Space ();
      morrow_point_data.target_storage = morrow_point.live_capacity;
      crystal_data.target_storage = crystal.live_capacity;

   ELSE IF powell.elevation > powell.minimum_power_pool_elevation + 20 THEN

      ratio = Current_Upper_Basin_Storage () / Upper_Basin_Total_Capacity ();

      flaming_gorge_data.target_storage =
               MAX (C_Volume_At (“flaming_gorge.rated_power_head_elevation”),
                     flaming_gorge.live_capacity * ratio);
      blue_mesa_data.target_storage =
                   MAX (C_Volume_At (“blue_mesa.rated_power_head_elevation”),
                         blue_mesa.live_capacity * ratio);
      navajo_data.target_storage =
                    MAX (C_Volume_At (“navajo.rated_power_head_elevation”),
                          navajo.live_capacity * ratio);
      morrow_point_data.target_storage = morrow_point.live_capacity;
      crystal_data.target_storage = crystal.live_capacity;

   ELSE

      flaming_gorge_data.target_storage =
                      C_Volume_At (“flaming_gorge.minimum_power_elevation”);
      blue_mesa_data.target_storage =
                         C_Volume_At (“blue_mesa.minimum_power_elevation”);
      navajo_data.target_storage =
                            C_Volume_At (“navajo.minimum_power_elevation”);
      morrow_point_data.target_storage =
                       C_Volume_At (“morrow_point.minimum_power_elevation”);
```

```
      crystal_data.target_storage =
                              C_Volume_At ("crystal.minimum_power_elevation");

   ENDIF

END


FUNCTION Flaming_Gorge_Target_Space ()

BEGIN

   IF AUGUST <= CURRENT_MONTH && CURRENT_MONTH <= DECEMBER THEN

      RETURN (target_space_data.flaming_gorge [CURRENT_MONTH]);

   ELSE

      remaining_avg_virgin_inflow =
         average_virgin_inflow_data.flaming_gorge [CURRENT_MONTH..JULY];

      remaining_actual_inflow = actual_inflow_data.reservoir [CURRENT_MONTH..
                                                              JULY];

      weighted_avg =
         weights_data.virgin [CURRENT_MONTH] * remaining_avg_virgin_inflow +
         weights_data.actual [CURRENT_MONTH] * remaining_actual_inflow;

      release_needed = (flaming_gorge.storage -
                        flaming_gorge.live_capacity +
                        weighted_avg) /
                       (AUGUST - CURRENT_MONTH);

      release_needed = MAX (release_needed, flaming_gorge.minimum_release);

      maximum_allowable_storage = flaming_gorge.storage -
                                  release_needed +
                                  green_river_below_fontenelle.inflow +
                                  above_fontenelle.exports +
                                  green_river_above_greendale_ut.gains;

      maximum_allowable_storage = MIN (maximum_allowable_storage,
                                       flaming_gorge.live_capacity);

      RETURN (flaming_gorge.live_capacity - maximum_allowable_storage);

   ENDIF

END


FUNCTION Blue_Mesa_Target_Space ()

BEGIN

   IF AUGUST <= CURRENT_MONTH && CURRENT_MONTH <= DECEMBER THEN

      RETURN (target_space_data.blue_mesa [CURRENT_MONTH]);
```

```
   ELSE

      remaining_avg_virgin_inflow =
         average_virgin_inflow_data.blue_mesa [CURRENT_MONTH..JULY];

      remaining_actual_inflow = actual_inflow_data.blue_mesa [CURRENT_MONTH..
                                                              JULY];

      weighted_avg =
         weights_data.virgin [CURRENT_MONTH] * remaining_avg_virgin_inflow +
         weights_data.actual [CURRENT_MONTH] * remaining_actual_inflow;

      release_needed = (blue_mesa.storage -
                        blue_mesa.live_capacity +
                        weighted_avg) /
                       (AUGUST - CURRENT_MONTH);

      release_needed = MAX (release_needed, blue_mesa.minimum_release);

      maximum_allowable_storage = blue_mesa.storage -
                                  release_needed +
                                  taylor_river_above_taylor_park.inflow +
                                  gunnison_river_above_blue_mesa.gains;

      maximum_allowable_storage = MIN (maximum_allowable_storage,
                                       blue_mesa.live_capacity);

      RETURN (blue_mesa.live_capacity - maximum_allowable_storage);

   ENDIF

END


FUNCTION Navajo_Target_Space ()

BEGIN

   IF AUGUST <= CURRENT_MONTH && CURRENT_MONTH <= DECEMBER THEN

      RETURN (target_space_data.navajo [CURRENT_MONTH]);

   ELSE

      remaining_avg_virgin_inflow =
         average_virgin_inflow_data.navajo [CURRENT_MONTH..JULY]

      remaining_actual_inflow = actual_inflow_data.navajo [CURRENT_MONTH..
                                                           JULY]

      weighted_avg =
         weights_data.virgin [CURRENT_MONTH] * remaining_avg_virgin_inflow +
         weights_data.actual [CURRENT_MONTH] * remaining_actual_inflow;

      release_needed = (navajo.storage -
                        navajo.live_capacity +
                        weighted_avg) /
                       (AUGUST - CURRENT_MONTH);
```

```
      release_needed = MAX (release_needed, navajo.minimum_release);

      maximum_allowable_storage = navajo.storage -
                                  release_needed +
                                  san_juan_river_near_archuleta_nm.flow +
                                  navajo_reservoir.demand;

      maximum_allowable_storage = MIN (maximum_allowable_storage,
                                    navajo.live_capacity);

      RETURN (navajo.live_capacity - maximum_allowable_storage);

   ENDIF

END


FUNCTION Current_Upper_Basin_Storage ()

BEGIN

   RETURN (flaming_gorge.storage +
           blue_mesa.storage +
           morrow_point.storage +
           crystal.storage +
           navajo.storage +
           powell.storage);

END


FUNCTION Upper_Basin_Total_Capacity ()

BEGIN

   RETURN (flaming_gorge.total_capacity +
           blue_mesa.total_capacity +
           morrow_point.total_capacity +
           crystal.total_capacity +
           navajo.total_capacity +
           powell.total_capacity);

END
```

D.3. Complex Lex/Yacc Version of Upper Basin Rule Curve Rule

```
RULE_NAME: powell_elevation
RULE_PRIORITY: 1
RULE_DEPENDENCIES: powell.elevation

BEGIN

   IF powell.elevation > powell.turbine_overload_elevation THEN

      flaming_gorge_data.target_storage = flaming_gorge.live_capacity -
                                          Target_Space (“flaming_gorge”);
      blue_mesa_data.target_storage = blue_mesa.live_capacity -
                                      Target_Space (“blue_mesa”);
      navajo_data.target_storage = navajo.live_capacity -
                                   Target_Space (“navajo”);
      morrow_point_data.target_storage = morrow_point.live_capacity;
      crystal_data.target_storage = crystal.live_capacity;

   ELSE IF powell.elevation > powell.rated_power_head THEN

      ratio = Current_Upper_Basin_Storage () / Upper_Basin_Total_Capacity ();

      flaming_gorge_data.target_storage = flaming_gorge.live_capacity -
         MAX (C_Volume_At (“flaming_gorge.rated_power_head_elevation”),
              flaming_gorge.live_capacity * ratio);
      blue_mesa_data.target_storage = blue_mesa.live_capacity -
                                      Target_Space (“blue_mesa”);
      navajo_data.target_storage = navajo.live_capacity -
                                   Target_Space (“navajo”);
      morrow_point_data.target_storage = morrow_point.live_capacity;
      crystal_data.target_storage = crystal.live_capacity;

   ELSE IF powell.elevation > powell.minimum_power_pool_elevation + 20 THEN

      ratio = Current_Upper_Basin_Storage () / Upper_Basin_Total_Capacity ();

      flaming_gorge_data.target_storage =
         MAX (C_Volume_At (“flaming_gorge.rated_power_head_elevation”),
              flaming_gorge.live_capacity * ratio);
      blue_mesa_data.target_storage =
         MAX (C_Volume_At (“blue_mesa.rated_power_head_elevation”),
              blue_mesa.live_capacity * ratio);
      navajo_data.target_storage =
         MAX (C_Volume_At (“navajo.rated_power_head_elevation”),
              navajo.live_capacity * ratio);
      morrow_point_data.target_storage = morrow_point.live_capacity;
      crystal_data.target_storage = crystal.live_capacity;

   ELSE

      flaming_gorge_data.target_storage =
         C_Volume_At (“flaming_gorge.minimum_power_elevation”);
      blue_mesa_data.target_storage =
         C_Volume_At (“blue_mesa.minimum_power_elevation”);
      navajo_data.target_storage =
         C_Volume_At (“navajo.minimum_power_elevation”);
      morrow_point_data.target_storage =
         Volume_At (“morrow_point.minimum_power_elevation”);
```

```
      crystal_data.target_storage =
         C_Volume_At ("crystal.minimum_power_elevation");

   ENDIF

END


FUNCTION Target_Space (reservoir)

BEGIN

   IF AUGUST <= CURRENT_MONTH && CURRENT_MONTH <= DECEMBER THEN

      RETURN (target_space_data.reservoir [CURRENT_MONTH]);

   ELSE

      remaining_avg_virgin_inflow =
         average_virgin_inflow_data.reservoir [CURRENT_MONTH..JULY];

      remaining_actual_inflow =
         actual_inflow_data.reservoir [CURRENT_MONTH..JULY];

      weighted_avg = weights_data.virgin [CURRENT_MONTH] *
                     remaining_avg_virgin_inflow +
                     weights_data.actual [CURRENT_MONTH] *
                     remaining_actual_inflow;

      release_needed = (reservoir.storage -
                        reservoir.live_capacity +
                        weighted_avg) /
                       (AUGUST - CURRENT_MONTH);

      release_needed = MAX (release_needed, reservoir.minimum_release);

      IF (reservoir == "flaming_gorge") THEN
         maximum_allowable_storage = flaming_gorge.storage -
                                     release_needed +
                                     green_river_below_fontenelle.inflow +
                                     above_fontenelle.exports +
                                     green_river_above_greendale_ut.gains;
      ELSE IF (reservoir == "blue_mesa") THEN
         maximum_allowable_storage = blue_mesa.storage -
                                     release_needed +
                                     taylor_river_above_taylor_park.inflow +
                                     gunnison_river_above_blue_mesa.gains;
      ELSE IF (reservoir == "navajo") THEN
         maximum_allowable_storage = navajo.storage -
                                     release_needed +
                                     san_juan_river_near_archuleta_nm.flow +
                                     navajo_reservoir.demand;
      ENDIF

      maximum_allowable_storage = MIN (maximum_allowable_storage,
                                       reservoir.live_capacity);

      RETURN (reservoir.live_capacity - maximum_allowable_storage);
```

```
   ENDIF

END


FUNCTION Current_Upper_Basin_Storage ()

BEGIN

   RETURN (flaming_gorge.storage +
           blue_mesa.storage +
           morrow_point.storage +
           crystal.storage +
           navajo.storage +
           powell.storage);

END


FUNCTION Upper_Basin_Total_Capacity ()

BEGIN

   RETURN (flaming_gorge.total_capacity +
           blue_mesa.total_capacity +
           morrow_point.total_capacity +
           crystal.total_capacity +
           navajo.total_capacity +
           powell.total_capacity);

END
```

D.4. CLIPS Version of Upper Basin Rule Curve Rule

```
;
; RULE powell_elevation
;
(defrule powell_elevation_1
    (declare (salience 1))
    (powell.elevation                  ?elevation)
    (powell.turbine_overload_elevation ?turbine_overload_elevation)
    (powell.rated_power_head           ?rated_power_head)
    (powell.minimum_pool_elevation     ?minimum_pool_elevation)
=>
    (if (> ?elevation ?turbine_overload_elevation) then
        (C_Set_Value “flaming_gorge.target_storage”
                     (- (C_Get_Value “flaming_gorge.live_capacity”)
                        (Target_Space “flaming_gorge”)))

        (C_Set_Value “blue_mesa.target_storage”
                     (- (C_Get_Value “blue_mesa.live_capacity”)
                        (Target_Space “blue_mesa”)))

        (C_Set_Value “navajo.target_storage”
                     (- (C_Get_Value “navajo.live_capacity”)
                        (Target_Space “navajo”)))

        (C_Set_Value “morrow_point.target_storage”
                     (C_Get_Value “morrow_point.live_capacity”))

        (C_Set_Value “crystal.target_storage”
                     (C_Get_Value “crystal.live_capacity”))

    else (if (> ?elevation ?rated_power_head)
        (C_Set_Value “flaming_gorge.target_storage”
                     (max (* (C_Get_Value “flaming_gorge.live_capacity”)
                             upper_basin_ratio)
                          (C_Volume_At “flaming_gorge.rated_power_head”)))

        (C_Set_Value “blue_mesa.target_storage”
                    (- (C_Get_Value “blue_mesa.live_capacity”)
                       (Target_Space “blue_mesa”)))

        (C_Set_Value “navajo.target_storage”
                    (- (C_Get_Value “navajo.live_capacity”)
                       (Target_Space “navajo”)))

        (C_Set_Value “morrow_point.target_storage”
                     (C_Get_Value “morrow_point.live_capacity”))

        (C_Set_Value “crystal.target_storage”
                     (C_Get_Value “crystal.live_capacity”))

      ratio = Current_Upper_Basin_Storage () / Upper_Basin_Total_Capacity ();

      flaming_gorge.target_storage =
         MAX (C_Volume_At (“flaming_gorge.rated_power_head_elevation”),
              flaming_gorge.live_capacity * ratio);

    else (if (> ?elevation ?minimum_pool_elevation)
        (C_Set_Value “flaming_gorge.target_storage”
```

```
                    (max (* (C_Get_Value “flaming_gorge.live_capacity”) ratio)
                        (C_Volume_At “flaming_gorge.rated_power_head”)))

        (C_Set_Value “blue_mesa.target_storage”
                    (max (* (C_Get_Value “blue_mesa.live_capacity”) ratio)
                         (C_Volume_At “blue_mesa.rated_power_head”)))

        (C_Set_Value “navajo.target_storage”
                    (max (* (C_Get_Value “navajo.live_capacity”) ratio)
                         (C_Volume_At “navajo.rated_power_head”)))

        (C_Set_Value “morrow_point.target_storage”
                    (C_Get_Value “morrow_point.live_capacity”)

        (C_Set_Value “crystal.target_storage”
                    (C_Get_Value “crystal.live_capacity”)

    else

        (C_Set_Value “flaming_gorge.target_storage”
                    (C_Volume_At “flaming_gorge.minimum_power_elevation”))

        (C_Set_Value “blue_mesa.target_storage”
                    (C_Volume_At “blue_mesa.minimum_power_elevation”))

        (C_Set_Value “navajo.target_storage”
                    (C_Volume_At “navajo.minimum_power_elevation”))

        (C_Set_Value “morrow_point.target_storage”
                    (C_Volume_At “morrow_point.minimum_power_elevation”))

        (C_Set_Value “crystal.target_storage”
                    (C_Volume_At “crystal.minimum_power_elevation”)))))
)


;
; FUNCTION Target_Space
;
(deffunction Target_Space (?reservoir)
    (if (> ?*current_month* ?august) then
        (return (C_Get_Value ?reservoir ?*current_month*)))

    (bind ?virgin_inflow 0)
    (bind ?month ?*current_month*)
    (while (>= ?month ?july) do
        (bind ?virgin_inflow (C_Get_Value “average_virgin_inflow” ?month))
        (bind ?month (+ ?month 1))
    )

    (bind ?actual_inflow 0)
    (bind ?month ?*current_month*)
    (while (>= ?month ?july) do
        (bind ?actual_inflow (C_Get_Value “actual_inflow” ?month))
        (bind ?month (+ ?month 1))
    )

    (bind ?virgin_weights (C_Get_Value “virgin” ?*current_month*))
    (bind ?actual_weights (C_Get_Value “actual” ?*current_month*))
```

```
    (bind ?predicted_inflow (+ (* ?virgin_weights ?virgin_inflow)
                               (* ?actual_weights ?actual_inflow)))

    (bind ?storage (C_Get_Value ?reservoir “storage”))
    (bind ?live_capacity (C_Get_Value ?reservoir “live_capacity”))

    (bind ?release_needed (/ (+ (- ?storage ?live_capacity)
                                ?predicted_inflow)
                             (- ?august ?*current_month*)))

    (bind ?minimum_release (C_Get_Value ?reservoir “minimum_release”))
    (bind ?release_needed (max ?release_needed ?minimum_release))

    (if (str-compare ?reservoir “flaming_gorge”) then
        (bind ?maximum_allowable_storage
              (- (+ ?storage
                    (C_Get_Value “green_river_below_fontenelle.inflow”)
                    (C_Get_Value “above_fontenelle.exports”)
                    (C_Get_Value “green_river_above_greendale_ut.gain”))
              ?release_needed))
    else (if (str-compare ?reservoir “blue_mesa”) then
        (bind ?maximum_allowable_storage
              (- (+ ?storage
                    (C_Get_Value “taylor_river_above_taylor_park.inflow”)
                    (C_Get_Value “gunnison_river_above_blue_mesa.gain”))
              ?release_needed))
    else (if (str-compare ?reservoir “navajo”) then
        (bind ?maximum_allowable_storage
              (- (+ ?storage
                    (C_Get_Value “san_juan_river_near_archuleta_nm.flow”)
                    (C_Get_Value “navajo_reservoir.demand”))
              ?release_needed)))))

    (bind ?maximum_allowable_storage (min ?maximum_allowable_storage
                                          ?live_capacity))

    (- ?live_capacity ?maximum_allowable_storage)
)


;
; FUNCTION upper_basin_ratio
;
(deffunction upper_basin_ratio ()
   (/ current_upper_basin_storage current_upper_total_capacity)
)


;
; FUNCTION current_upper_basin_storage
;
(deffunction current_upper_basin_storage ()
   (+ (C_Get_Value “flaming_gorge.storage”)
      (C_Get_Value “blue_mesa.storage”)
      (C_Get_Value “morrow_point.storage”)
      (C_Get_Value “crystal.storage”)
      (C_Get_Value “navajo.storage”)
      (C_Get_Value “powell.storage”)
```

```
   )
)


;
; FUNCTION current_upper_total_capacity
;
(deffunction current_upper_total_capacity ()
   (+ (C_Get_Value “flaming_gorge.total_capacity”)
      (C_Get_Value “blue_mesa.total_capacity”)
      (C_Get_Value “morrow_point.total_capacity”)
      (C_Get_Value “crystal.total_capacity”)
      (C_Get_Value “navajo.total_capacity”)
      (C_Get_Value “powell.total_capacity”)
   )
)
```

D.5. Tcl Version of Upper Basin Rule Curve Rule

```
RULE_NAME: powell_elevation
RULE_PRIORITY: 1
RULE_DEPENDENCIES: powell.elevation

BEGIN

   set powell.elevation             [C_Get_Value powell.elevation]
   set powell.turbine_overload_elev [C_Get_Value
                                          powell.turbine_overload_elev]
   set powell.rated_power_head      [C_Get_Value powell.rated_power_head]
   set powell.min_power_pool_elev   [C_Get_Value powell.min_power_pool_elev]

   if {$powell.elevation > $powell.turbine_overload_elev} {

      C_Set_Value flaming_gorge_data.target_storage
                  [Maximize_Target_Storage flaming_gorge]
      C_Set_Value blue_mesa_data.target_storage
                  [Maximize_Target_Storage blue_mesa]
      C_Set_Value navajo_data.target_storage
                  [Maximize_Target_Storage navajo]
      C_Set_Value morrow_point_data.target_storage
                  [C_Get_Value morrow_point.live_capacity]
      C_Set_Value crystal_data.target_storage
                  [C_Get_Value crystal.live_capacity]

   } elseif {$powell.elevation > $powell.rated_power_head} {

      C_Set_Value flaming_gorge_data.target_storage
                  [Almost_Maximize_Target_Storage flaming_gorge]
      C_Set_Value blue_mesa_data.target_storage
                  [Maximize_Target_Storage blue_mesa]
      C_Set_Value navajo_data.target_storage
                  [Maximize_Target_Storage navajo]
      C_Set_Value morrow_point_data.target_storage
                  [C_Get_Value morrow_point.live_capacity]
      C_Set_Value crystal_data.target_storage
                  [C_Get_Value crystal.live_capacity]

   } elseif {$powell.elevation > [expr $powell.min_power_pool_elev + 20]} {

      C_Set_Value flaming_gorge_data.target_storage
                  [Almost_Maximize_Target_Storage flaming_gorge]
      C_Set_Value blue_mesa_data.target_storage
                  [Almost_Maximize_Target_Storage blue_mesa]
      C_Set_Value navajo_data.target_storage
                  [Almost_Maximize_Target_Storage navajo]
      C_Set_Value morrow_point_data.target_storage
                  [C_Get_Value morrow_point.live_capacity]
      C_Set_Value crystal_data.target_storage
                  [C_Get_Value crystal.live_capacity]

   } else {

      C_Set_Value flaming_gorge_data.target_storage
                  [C_Volume_At flaming_gorge.minimum_power_elevation]
      C_Set_Value blue_mesa_data.target_storage
                  [C_Volume_At blue_mesa.minimum_power_elevation]
```

```
      C_Set_Value navajo_data.target_storage
                  [C_Volume_At navajo.minimum_power_elevation]
      C_Set_Value morrow_point_data.target_storage
                  [C_Volume_At morrow_point.minimum_power_elevation]
      C_Set_Value crystal_data.target_storage
                  [C_Volume_At crystal.minimum_power_elevation]
   }

END


proc Target_Space {reservoir} {

   if {$AUGUST <= $CURRENT_MONTH && $CURRENT_MONTH <= $DECEMBER} {

      return [C_Get_Value target_space_data.$reservoir($CURRENT_MONTH)]

   } else {

      set remaining_avg_virgin_inflow [C_Get_Value \
                average_virgin_inflow_data.$reservoir($CURRENT_MONTH..$JULY)]

      set remaining_actual_inflow \
          [C_Get_Value actual_inflow_data.$reservoir($CURRENT_MONTH..$JULY)]

      set weighted_avg \
          [expr [C_Get_Value weights_data.virgin($CURRENT_MONTH)] * \
                $remaining_avg_virgin_inflow + \
                [C_Get_Value weights_data.actual($CURRENT_MONTH)] * \
                $remaining_actual_inflow]

      set release_needed [expr ([C_Get_Value $reservoir.storage] - \
                                [C_Get_Value $reservoir.live_capacity] + \
                                $weighted_avg) / \
                                ($AUGUST - $CURRENT_MONTH)]

      set release_needed \
          [MAX $release_needed [C_Get_Value $reservoir.minimum_release]]

      if {$reservoir == flaming_gorge} {
         set maximum_allowable_storage \
             [expr [C_Get_Value flaming_gorge.storage] - \
                   $release_needed + \
                   [C_Get_Value green_river_below_fontenelle.inflow + \
                   [C_Get_Value above_fontenelle.exports + \
                   [C_Get_Value green_river_above_greendale_ut.gains]]
      } elseif {$reservoir == blue_mesa} {
         set maximum_allowable_storage \
             [expr [C_Get_Value blue_mesa.storage] - \
                   $release_needed + \
                   [C_Get_Value taylor_river_above_taylor_park.inflow] + \
                   [C_Get_Value gunnison_river_above_blue_mesa.gains]]
      } elseif {$reservoir == navajo} {
         set maximum_allowable_storage \
             [expr [C_Get_Value navajo.storage] - \
                   $release_needed + \
                   [C_Get_Value san_juan_river_near_archuleta_nm.flow] + \
                   [C_Get_Value navajo_reservoir.demand]]
      }
```

```
      set maximum_allowable_storage \
          [expr [MIN $maximum_allowable_storage $reservoir.live_capacity]]

      return [expr $reservoir.live_capacity - $maximum_allowable_storage]
}


proc Upper_Basin_Storage_Ratio {} {

   set storage [Current_Upper_Basin_Storage]
   set capacity [Upper_Basin_Total_Capacity]

   return [expr $storage / $capacity]
}


proc Current_Upper_Basin_Storage {} {

   set fg [C_Get_Value flaming_gorge.storage]
   set bm [C_Get_Value blue_mesa.storage]
   set mp [C_Get_Value morrow_point.storage]
   set cr [C_Get_Value crystal.storage]
   set nv [C_Get_Value navajo.storage]
   set pw [C_Get_Value powell.storage]

   return [expr $fg + $bm + $mp + $cr + $nv + $pw]
}


proc Upper_Basin_Total_Capacity {} {

   set fg [C_Get_Value flaming_gorge.total_capacity]
   set bm [C_Get_Value blue_mesa.total_capacity]
   set mp [C_Get_Value morrow_point.total_capacity]
   set cr [C_Get_Value crystal.total_capacity]
   set nv [C_Get_Value navajo.total_capacity]
   set pw [C_Get_Value powell.total_capacity]

   return [expr $fg + $bm + $mp + $cr + $nv + $pw]
}


proc Maximize_Target_Storage {site} {

   set value1 [C_Get_Value $site.live_capacity]
   set value2 [Target_Space $site]

   return [expr $value1 + $value2]
}


proc Almost_Maximize_Target_Storage {site} {

   set ratio         [Upper_Basin_Storage_Ratio]
   set live_capacity [C_Get_Value $site.live_capacity]
   set volume        [C_Volume_At $site.rated_power_head_elevation]
   set max           [Max $volume [expr $live_capacity * $ratio]]
```

```
    return [expr $live_capacity - $max]
}
```

Appendix E

C/Tcl Routines

E.1. Value Access Routines

These subroutines are used to read and write values in the *RiverWare* model. Many have date/times as arguments, which are used to specify when during the simulation the value or values need to be read or written. They are also used to convert the read or written values to monthly units. If no date/time is specified, the current simulation date/time is used. For those subroutines that need to return a value, a rule execution exception will occur if no value can be returned.

E.1.1. Read

C_GetValue

`object.slot scale units <dateTime>`

Return the value of the object.slot in the specified scale and units at the specified date/time.

C_GetValueNaN

`object.slot scale units <dateTime>`

The same as **C_GetValue**, except that no rule execution exception will occur if there is no value (i.e., NaN) for the object.slot at the specified date/time.

C_GetList

`object.slot scale units timestep startDateTime endDateTime`

Return a list of values from the object.slot in the specified scale and units over the specified date/time range using the specified timestep increment.

C_GetTableValue

`object.slot scale units row column <dateTime>`

Return the value of the table object.slot in the specified scale and units for the specified row and column.

C_GetTableRowList

`object.slot <scale units> row startColumn endColumn <dateTime>`

Return a list of values from the table object.slot in the specified scale and units for the specified row and columns.

C_GetTableColumnList

`object.slot <scale units> column startRow endRow <dateTime>`

Return a list of values from the table object.slot in the specified scale and units for the specified rows and column.

C_SumList

```
object.slot scale units timestep startDateTime endDateTime
```

Return the summation of a list of values from the object.slot in the specified scale and units over the specified date/time range using the specified timestep increment.

C_SumTableRowList

```
object.slot <scale units> row startColumn endColumn <dateTime>
```

Return the summation of a list of values from the table object.slot in the specified scale and units for the specified row and columns.

C_SumTableColumnList

```
object.slot <scale units> column startRow endRow <dateTime>
```

Return the summation of a list of values from the table object.slot in the specified scale and units for the specified rows and column.

C_SumObjectList

```
objectList slot scale units <dateTime>
```

Return the summation of values from a list of objects for a specified slot in the specified scale and units.

E.1.2. Write

C_SetValue

```
object.slot scale units <dateTime> value
```

Write the value to the object.slot in the specified scale and units.

C_SetList

```
object.slot scale units timestep startDateTime valueList
```

Write the list of values to the object.slot in the specified scale and units using the timestep as an increment.

C_CopySlot

```
destination.slot source.slot <dateTime>
```

Copy the value from the source object.slot to the destination object.slot.

C_CopySlots

```
destination.slot source.slot timestep startDateTime endDateTime
```

Copy the values from the source object.slot to the destination object.slot from using the timestep as an increment.

C_ConstrainSlot

```
object.slot operator constraint.slot <dateTime>
```

Constrain the value of object.slot to the minimum of its value and the value of constraint.slot if operator is “<=”. Constrain the value of object.slot to the maximum of its value and the value of constraint.slot if operator is “>=”.

C_ConstrainSlot

```
object.slot operator constraintValue <dateTime>
```

Constrain the value of object.slot to the minimum of its value and the value of constraintValue if operator is "<=". Constrain the value of object.slot to the maximum of its value and the value of constraintValue if operator is ">=".

E.1.3. Test

C_IsSlotValueNaN

```
object.slot <dateTime>
```

Return true if the value of object.slot at the specified date/time has not been set.

C_IsNaN

```
value
```

Return true if the value specified is NaN.

E.2. Object-specific Routines

These subroutines are used to perform mass balance operations using the provided values. All have date/times as arguments, which are used to specify when during the simulation the values need to be read. They are also used to convert the read or written values to monthly units. If no date/time is specified, the current simulation date/time is used. Returned values are converted to the specified scale and units.

C_SolveInflow

```
reservoir outflow outflowScale outflowUnits storage
storageScale storageUnits <prevStorage prevStorageScale
prevStorageUnits> inflowScale inflowUnits <date_time>
```

Solve for inflow for the specified reservoir using the provided outflow, storage and previous storage values. If previous storage is not specified, the reservoir's previous storage value is used.

C_SolveOutflow

```
reservoir inflow inflowScale inflowUnits storage storageScale
storageUnits <prevStorage prevStorageScale prevStorageUnits>
outflowScale outflowUnits <date_time>
```

Solve for outflow for the specified reservoir using the provided inflow, storage and previous storage values. If previous storage is not specified, the reservoir's previous storage value is used.

C_SolveStorage

```
reservoir inflow inflowScale inflowUnits outflow outflowScale
outflowUnits <prevStorage prevStorageScale prevStorageUnits>
storageScale storageUnits <date_time>
```

Solve for storage for the specified reservoir using the provided inflow, outflow and previous storage values. If previous storage is not specified, the reservoir's previous storage value is used.

C_NetNonShortDivReq

```
aggDiversion scale units <date_time>
```

Return the net non-short diversion requirement for the aggregate diversion.

E.3. Sub-basin Routines

These subroutines are used to perform operations on sub-basins and return one or more values. Some have date/times as arguments, which are used to convert the returned values to monthly units. Omitted date/times default to the current simulation date/time. Returned values are converted to the specified scale and units.

C_GetAllNamedBasins

Return a list of all named sub-basins.

C_GetObjectsInBasin

```
basinDesignator
```

Return a list of all objects in the named basin.

C_AggOverTime

```
basinDesignator slotName AggFunc AggFilt scale units <startDate
<endDate>>
```

Extract and return a list of values from a sub-basin based on a provided slot over a date/time range. The aggregation function (“SUM”, “AVG”, “MIN”, “MAX”) and the aggregation filter (“IN/OUT”, “IN”, “OUT”) are used to determine how the values are grouped for each timestep and which values are used in the grouping, respectively.

C_AggOverObj

```
basinDesignator slotName AggFunc AggFilt scale units <startDate
<endDate>>
```

Extract and return a list of values from a sub-basin based on a provided slot over a date/time range. The aggregation function (“SUM”, “AVG”, “MIN”, “MAX”) and the aggregation filter (“IN/OUT”, “IN”, “OUT”) are used to determine how the values are grouped for each object.slot and which values are used in the grouping, respectively.

C_AggOverTimeObj

```
basinDesignator slotName AggFunc1 aggFunc2 AggFilt scale units
<startDate <endDate>>
```

Extract and return a value from a sub-basin based on a provided slot over a date/time range. The aggregation functions (“SUM”, “AVG”, “MIN”, “MAX”) and the aggregation filter (“IN/OUT”, “IN”, “OUT”) are used to determine how the values are grouped for each timestep/object.slot and which values are used in the grouping, respectively.

C_AggOverObjTime

```
basinDesignator slotName AggFunc1 aggFunc2 AggFilt scale units
<startDate <endDate>>
```

Extract and return a value from a sub-basin based on a provided slot over a date/time range. The aggregation functions (“SUM”, “AVG”, “MIN”, “MAX”) and the aggregation filter (“IN/OUT”, “IN”, “OUT”) are used to determine how the values are grouped for each object.slot/timestep and which values are used in the grouping, respectively.

E.4. Conversion Routines

These subroutines are used to convert values from one scale and units to another. Many have date/times as arguments, which are used to convert the returned values to monthly units. If no date/time is specified, the current simulation date/time is used.

C_TableInterpolate

```
object.slot fromColumn fromValue fromScale fromUnits toColumn
toScale toUnits
```

Return a value stored in one column (i.e., "toColumn") of a table based on the value stored in a neighboring column on the same table.

C_StorageToArea

```
reservoir storage storageScale storageUnits areaScale areaUnits
<dateTime>
```

Convert storage to area for the specified reservoir.

C_ElevationToArea

```
reservoir elevation elevationScale elevationUnits areaScale
areaUnits <dateTime>
```

Convert elevation to area for the specified reservoir.

C_ElevationToStorage

```
reservoir elevation elevationScale elevationUnits storageScale
storageUnits <dateTime>
```

Convert elevation to storage for the specified reservoir.

C_ElevationToMaxRegulatedSpill

```
reservoir elevation elevationScale elevationUnits spillScale
spillUnits <dateTime>
```

Convert elevation to maximum regulated spill for the specified reservoir.

C_ElevationToUnregulatedSpill

```
reservoir elevation elevationScale elevationUnits spillScale
spillUnits <dateTime>
```

Convert elevation to unregulated spill for the specified reservoir.

C_OperatingHeadToMaxRelease

```
reservoir elevation elevationScale elevationUnits releaseScale
releaseUnits <dateTime>
```

Convert operating head (i.e., elevation) to maximum release for the specified reservoir.

C_FlowToVolume

```
flowObject.slot {volumeScale volumeUnits} <dateTime>
```

Convert the provided flow to a volume using the provided scale and units.

C_VolumeToFlow

`volumeObject.slot {flowScale flowUnits} <dateTime>`

Convert the provided volume to a flow using the provided scale and units.

C_ConvertValue

`fromValue fromScale fromUnits toScale toUnits <dateTime>`

Convert the provided value from the “from” scale and units to the “to” scale and units.

E.5. Date and Time Routines

These subroutines return or manipulate previously returned information related to the current *RiverWare* simulation clock.

C_GetStartDateTime

Return the date and time at the simulation’s start.

C_GetStartYear

Return the year at the simulation’s start.

C_GetStartMonth

Return the month at the simulation’s start.

C_GetStartDay

Return the month day at the simulation’s start.

C_GetStartHour

Return the hour at the simulation’s start.

C_GetStartMinute

Return the minute at the simulation’s start.

C_GetStartSecond

Return the second at the simulation’s start.

C_GetEndDateTime

Return the date and time at the simulation’s end.

C_GetEndYear

Return the year at the simulation’s end.

C_GetEndMonth

Return the month at the simulation’s end.

C_GetEndDay

Return the month day at the simulation’s end.

C_GetEndHour

Return the hour at the simulation's end.

C_GetEndMinute

Return the minute at the simulation's end.

C_GetEndSecond

Return the second at the simulation's end.

C_GetDateTime

Return the current simulation date and time.

C_GetYear

Return the current simulation year.

C_GetMonth

Return the current simulation month.

C_GetDay

Return the current simulation month day.

C_GetHour

Return the current simulation hour.

C_GetMinute

Return the current simulation minute.

C_GetSecond

Return the current simulation seconds.

C_IncrYear

`dateTime <number>`

Increment the provided date/time's year by the number specified.

C_IncrMonth

`dateTime <number>`

Increment the provided date/time's month by the number specified.

C_IncrDay

`dateTime <number>`

Increment the provided date/time's month day by the number specified.

C_IncrHour

`dateTime <number>`

Increment the provided date/time's hour by the number specified.

C_IncrMinute

`dateTime <number>`

Increment the provided date/time's minute by the number specified.

C_IncrSecond

`dateTime <number>`

Increment the provided date/time's second by the number specified.

C_DecrYear

`dateTime <number>`

Decrement the provided date/time's year by the number specified.

C_DecrMonth

`dateTime <number>`

Decrement the provided date/time's month by the number specified.

C_DecrDay

`dateTime <number>`

Decrement the provided date/time's month day by the number specified.

C_DecrHour

`dateTime <number>`

Decrement the provided date/time's hour by the number specified.

C_DecrMinute

`dateTime <number>`

Decrement the provided date/time's minute by the number specified.

C_DecrSecond

`dateTime <number>`

Decrement the provided date/time's seconds by the number specified.

C_GetDaysInMonth

`dateTime`

Return the number of days in the month for the provided date/time.

C_GetMonthFromDate

`dateTime`

Return the month usng the provided date/time.

C_GetStepSeconds

`<dateTime>`

Return the number of seconds for the simulation's current timestep.

E.6. Random Deviation Routines

These subroutines are included to allow the USBR to duplicate *CRSS* results.

C_ResetRanDev

`"rewind" | "no_rewind" startYear`

Reset the random deviation function.

C_RanDev

Return a "random" number from a list of pre-generated random numbers used by *CRSS*.

E.7. Utility Routines

These subroutines provide general utility functionality.

C_Exit

`message`

Exit the simulation and display the provided message.

C_Min

`value1 <value2 ...>`

Return the minimum value from the provided numbers.

C_Min

`valueList`

Return the minimum value from the provided list of numbers.

C_Max

`value1 <value2 ...>`

Return the maximum value from the provided numbers.

C_Max

`valueList`

Return the maximum value from the provided list of numbers.

E.8. Output Routines

These subroutines provide output functionality.

C_Print

`<message>`

Write the printed message to the system's debug buffer.

C_Trace

`traceLevel <message>`

Write the printed message to the system's debug buffer if the specified trace level is turned on.

E.9. Rule-specific Routines

These subroutines provide functionality specific to USBR rules.

`C_Six02a`

Calculate the value for 602a.

`C_CriticalPeriodUBDepletions`

`criticalPeriod`

Calculate the value for critical period upper basin depletion based on the provided critical period.

`C_TargetSpace`

`reservoir`

Calculate the target space for the provided reservoir.

`C_MaxAllowableStorage`

`reservoir`

Calculate the maximum allowable storage for the provided reservoir.

`C_ReleaseNeeded`

`reservoir`

Calculate the needed release for the provided reservoir.

`C_InflowAbove`

`reservoir`

Calculate the inflow above the provided reservoir.

`C_ConsumptionAbove`

`reservoir`

Calculate the consumption above the provided reservoir.

`C_PredictedInflow`

`reservoir`

Calculate the predicted inflow for the provided reservoir.

E.10. Key

<optional>	parameter or parameters are optional
“literal”	literal string is necessary
|	or indicator

Appendix F

User Test 2

F.1. Pre-User Testing

F.1.1. Preparation

UserE was familiar with the language and the editor and was working on turning the flowchart-based logic sent by the San Juan folks into rules and functions which could be used in the rule language. Much of the logic of the rules and how it could used in *RiverWare* had been completed and I was asked to assist in the construction of the rules/ functions.

In order to determine what the rules did, UserE first translated them into procedural code. Since the logic consisted primarily of simple and nested conditionals, UserE thought that structuring the rules in a manner that was similar to how one would code C was acceptable. As the rule language is not structured this way, we had to restructure the logic in order to fit it into the rule language framework. Initially, this did not seem intuitive to UserE, although after showing how one of the slot value settings could be extracted. It was relatively easy to remove the others. In doing so, this made it more clear what had to be true in order to set each given slot. This had the drawback that it separated the logic, so that changes in one place might result in the need for changes elsewhere.

As this process went along, it became clear that there was a misunderstanding regarding the setting of values. The language currently only supports the setting of slot values. There is no mechanism in place to support setting of local variable in rules. Yet, UserE thought that this was possible and was structuring the rules to do this. When I pointed this out, there was some concern that the decision criteria needed to determine what values a slot should be set to would have to be duplicated. I suggested the use of functions as an alternative and UserE found that acceptable. Yet, the use of functions has its own drawbacks in that the same function may be executed many times without having its inputs (and thus outputs) changed. Yet, the function will be executed multiple times which may greatly impact the performance of the rule's execution. This is not a new problem in functional languages, yet it still one which must be addressed, since performance is an issue.

Once it became clear how the rules and functions needed to be constructed and combined, the process of completing the ruleset went very smoothly.

A number of issues came out in the pre-user and user testing. These include issues related to the language as well as issues related to usability of the environment. In addition, there were bugs that became apparent. For now, the bugs will be ignored insofar as they do not relate to either language design or environment usability.

UserA was tasked with developing a ruleset for the San Juan river basin in advance of a group that would be coming in for training. He is a civil engineer who is very familiar with *RiverWare* and has a good understanding of the Tcl-based rule system. I did not work directly with him while he was creating the ruleset, but would talk to him every day or so in order to help him out and to access his progress.

His feedback was generally favorable and he found the structure editor to be intuitive. He did, however,

come up with a number of problems that were related to the interface.

- Quintuple mouse click The structure editor allows for limited in-line editing of expressions. This is to allow a user to enter numbers, strings, etc. directly into the expressions without having to provide for some explicit alternative. This seemed like a good solution in that it does not clutter up the palette with a lot of unnecessary options and allows users some flexibility with respect to how they create/ edit their rules and functions.
  The way in-line editing works is that a user may double click on a terminal expression and have an editable text field containing the current expression pop-up over the double-clicked expression. The user may then enter any text s/he wishes into this field and upon hitting a carriage return have the information parsed and checked for syntactic correctness. In this manner, the user may enter terminal expressions, such as numbers, and complex expressions, such as binary expressions. Whatever is entered is parsed and checked that it is syntactically correct and that its type matches the expected type for the expression for which it is substituting.

  As described, this editing mechanism works well. The problem that arose dealt with the way in which a user would typically use in-line editing. As stated above, in order to initiate in-line editing, the user must double click on the expression to be edited. Although a menu item is provided for the double click impaired, the double click is likely to be the standard way of initiating in-line editing. Now, once the in-line editor is invoked, the user must either triple click to select the entire contents of the line (a double click will work in rare cases) or click and drag to select the entire line. Since this is generally what the user will want to do, the user is required to perform a quintuple click in order to get an expression ready for editing. Again, there are other options, although these have other drawbacks which make them undesirable.

  UserA complained that his hand was getting very tired from all the clicking. Since the user generally wanted to edit the entire expression when doing in-line editing, it seems appropriate to automatically select the entire expression each time in-line editing was initiated. This will save the user 60% of their clicks in most cases, although there are certainly causes in which the user may not wish to edit the entire expression, but only change a portion of it. In these cases, the user is required to perform clicks in additional to the double click in order to select the portion of the expression they wish to edit. In addition, there is the potential for user error if s/he inadvertently types something while the entire expression is highlighted. In this case, the user will lose the original expression. In this case, the user can simple click outside the in-line editor text field to cancel in-line editing and return the original expression.

- Long line length The structure editor formats the expressions for the user as the expressions are built up. The user has no control over this. This has been known to be a problem for some time, but there have been other higher priority tasks to attend to.
  One known consequence of this “feature” is that long expressions can get lost off the right side of the editor. This is mitigated somewhat by the use of scroll bars, but this is a poor solution. There are two alternatives. One is to determine an optimum line length for the whole expression and insert carriage returns, as appropriate to make sure these line lengths are never violated. This has the drawback that it could insert a carriage return at a place which might obscure the meaning of the expression. While it may be possible to come up with some rules which make this unlikely, there are still so many unknowns (including lengths of terminal expressions and preferences of users) that the second alternative seems better. This approach is to allow the user to explicitly insert carriage returns between expressions and thus determine the appropriate line length and best layout for their needs.

  While this solution does give the user some flexibility and allows two syntactically identical expressions to have a different layout, I do not believe this is a serious drawback (if it is a drawback at all). The intent of the structure editor is to provide them with syntactic guidance rather than mandating all aspects of appearance. And, at any rate, this is not possible since the user has the ability to name groups, rules, and functions with few constraints.

- Parentheses when new left child is created. When an expression is added, mandatory parentheses are added as appropriate. These parentheses are not strictly needed from an internal expressio2n standpoint since the expression tree serves to disambiguate the meaning of the expression. Yet, since the underlying expression tree is not explicitly visible to the user, these parentheses make the expression easier to read and interpret. As a side note, it is possible to determine the structure of the underlying expression tree by selecting operators and seeing which sub-expressions are selected, yet this is not a good way of seeing the tree and does not help if the expressions are not in an interactive environment. Thus, these parentheses serve to disambiguate the expression's meaning for the user and, as it turns out, for the parser. The latter is true since the parentheses need to be written with the expression so that the parser will know the precedence order of the expressions.
  A problem occurs when an existing expression becomes the LHS of a new binary expression. In these cases, I wasn't adding parentheses. While this is a bug, it was one that could easily be overlooked and not be noticed until the ruleset was saved and re-loaded. This has been fixed.

- Functions whose type is not yet known. The function editor tries to be intelligent with respect to the return type of the function. Based on the function's body, it is able to determine the return type. This works well in many situations and frees the user up from having to specify the return type explicitly and for the editor from having to check that the user-specified return type matches the body's type. Yet, this comes into conflict with another goal of the structure editor. Currently, the structure editor only allows the user to substitute expressions which have a type which is appropriate to the context. Thus, if an expression is NUMERIC, only a function (or some other NUMERIC expression) which returns a number can be placed there.
  Yet, if the user wishes to create a function without specifying it body, this function will have a return type of ERROR. As such, it will not be allowed in any expression. Thus, the user must either put in a place holder expression (say a 1.0 if the function is to be NUMERIC) or work from bottom up. Both of these options seem sub-optimal. The first seems like an unnecessary inconvenience, although would work, I suppose. As for the bottom-up approach to coding, I am not convinced that all users work this way and moreover should be compelled to work this way. It seems that, to the extent possible, a user should be able to work either top-down or bottom-up. As it turns out (and as will be discussed later) the structure editor will impose some restrictions on the users' ability to do strictly top-down development.

  A good solution to the above problem, which is really akin the first approach, is to allow a return type to be explicitly selected on the function editor. This return type will be set automatically as the user creates the function, yet the editor will allow the user to specify the return type without necessarily having to specify a complete function. As mentioned above, the editor will have to check to insure that the user-specified return type matches the function's return type, but since the editor can already determine the function's return type, this should not be difficult.

  The only potential drawback is if the user wishes to change to return type of the entire function. For if the function has a return type, any expression that is added to the function that is of a different type will not match and therefore be rejected by the editor. A solution to this is to allow for a new function type if the user is replacing the entire function body and perform return type checks otherwise. The latter case might seem to be not very useful, since if a user has an ?+?, this is assumed to be NUMERIC. Yet, when a IF expression is placed in the function body, the return type of the function will not be known until one of the IF's consequences is known.

F.2. User Test

F.2.1. Initial Training

The user testing took place as part of a *RiverWare* training course. This training course was specific to the rulebased simulation component of *RiverWare*, although by necessity touched on many of the general

aspects of *RiverWare*. The users' goal was to learn how rulebased simulation worked, learn about the rule language/editor and how to use it, and to apply what they were to learn to a particular problem which they needed to solve.

In order to properly prepare for this training course, we requested and received the specifications for the problem they needed to work on. It was not written in a way the matched the rule language, although it could be translated. Our intent was to prepare a model/ruleset that corresponded to their system and test out both the model and rule language/editor to insure that they could effectively work on their problem while they were here. See above for information on this process.

User testing took place over a three day period. On the first day, users went through a simple rulebased simulation tutorial. The model was the simple two reservoir model and the ruleset was the simple three rule model. The rules had been previously written in the new rule language and they opened and viewed the rules. They did not edit the rules, although they did toggle the activeness of two of the rules. They did run simulations and view results. This tutorial took about two hours. I did not participate in this part of the user training.

On the second day, the new rule language itself was introduced. We started the day with a 1.25 hour session in which we presented the language. Although we had originally planned for Edie to give this talk, she decided that she didn't want to do it and, at the last minute, asked me to do it. As such, this talk was not prepared in advance. In this talk, I described the languages overall structure (i.e., rulesets, groups, rules, functions) and how these parts fit together to make up a set of policy rules. I described the languages types (i.e., NUMERIC, BOOLEAN, etc.) and the languages statements (i.e., rule statements and expression statements). In addition, I showed how to write some simple rules using the language. I used a procedural language (like FORTRAN) to show how the language related to it and to show how similar statements might be expressed. This session went well and there was not much feedback.

We took a 20 minute break.

10:20am After the break, we came back to show how a subset of their rules could be expressed in the new rule language. Edie led this session. We first presented them with this subset in a procedural form. Although this was similar to what they had given us, Edie had cleaned it up and put into a clean, procedural structure. We then started by highlighting and defining functions that were in the rules.

10:35am While we were describing the functional nature, referential transparency and order independence of the rule statements of the language, User2 mentioned that he thought this was similar to a spreadsheet.

Continued to talk through writing of rules using the procedural code as an example.

10:50am Needed to create a function for commonly used logic even though it was originally the RHS of a rule. To a degree, this made this rule statement a little more difficult to read, since the logic for it was now obscured. On the other hand, since this logic was used in more than one rule statement, it made the rule's policy as a whole more understandable, since it encapsulated the logic. Of course, the users of this new rule language are not required to encapsulate logic in this manner.

11:00am User1 & User3 feel that flowcharts (even when using a functional paradigm) are easier to explain to management level engineers, non-computer programmers and maybe even other users. This is an interesting point to consider and one which I would like to eventually determine if its true. [Alternate wording: Users felt that flowcharts were a preferable medium for communicating their policies to others.]

11:15am User1 mentioned that he feels he would have to write out code procedurally first and then convert to the more functional structure of the new rule language.

11:30am Converting one rule statement to the new rule language (Blanco0 generated a considerable amount of discussion regarding the correctness of the logic).

11:50am User2: “This a case for organization. Even with a small set of statements, we get into water dependencies which can make it easy to loose track of what’s going on.” They seemed to feel that the logic was probably okay, but were not entirely sure. They decided to write out the rule statement in order to see the logic and then decide if this was correct. As they were writing the rule, they decided that is was correct and confirmed this once it was finished.

My observation here was that since the procedural code combined the logic for all three rule statements (each of which set one diversion) into one, they did not really think about the individual logic for the setting of each diversion. As such, when this logic was removed from the logic used to set the other diversions, it became unclear to them whether the logic was correct. It is unclear whether this is good thing or a bad thing. In both cases, the underlying logic of the ruleset can be obscured. In the case where they take the ruleset as a whole, they have the benefit of grouping the similar logic explicitly. This helps them maintain how different decisions relate, yet can obscure each decision. If the ruleset is broken up, each decision is more clear, but it is probably less clear how the decisions relate, even if using subroutines in a disciplined manner.

12:00pm User1: “This is nifty”, since we can use functions with different rule statements. He felt that this would allow him to write one set of rule statements and then create different functions which would work either upstream or downstream in order to test different policies without making changes to the base rules.

12:05pm User1: “This is really interesting - could take some getting used to.”

We didn’t go into the logic of all the functions that could be created for this partial ruleset. We only discussed those that were used as part of more than one rule statements. This was to conserve time.

12:05pm One option we mentioned to them was that it was possible to split this one rule (with three rule statements) into three rules. They weren’t immediately sure what the effect of this would be and whether it was even necessary, given that they weren’t sure that they could assign meaningful priorities to the resulting rules. They decided that they needed a better idea of how this current system worked before they could decide.

12:15pm Concluding remarks: User1: “Looks neat.” User2: “Good stuff. Could be tough to get it done.” User3: “I like it, I’m looking forward to using it.”

12:15pm User2 seemed to find moving Blanco to rule statement confusing because he didn’t seem to understand the underlying logic and there found it difficult to express using a rule statement. I suspect he would have the same problem with a procedural language as well and that it might be good that they logic was separated and that he had to understand and be able to express it in isolation.

After lunch (2pm), the users started to write the portion of the ruleset which we had discussed in the previous session. I informed them that I would give them as little help as I thought was reasonable for them to get their job them, while still being able to observe what seems intuitive and what was difficult to understand. I also asked them to “think out loud” while they worked and to vocalize any questions or comments they had while working.

The four users were split into two groups. I worked with User3 and User4 while User1 and User2 spent time working on a more detailed look into the entire policy for the ruleset. I sat between User3 and User4 (and later User1 and User2) while they worked on the creating the rules and functions in this subset. I tried to pay attention to both as much as possible during this test.

F.2.2. User3 and User4

2:05pm User4 creates a new ruleset using the "New Ruleset" option on the menu. He asks, "Do I need to give a name to the ruleset?"

He doesn't at this point, since it is implicitly named "NoName" and he will be asked for a name when he saves the ruleset. I am not sure this is really an issue, since this kind of unnamed editing entity is used elsewhere (Word, FrameMaker, etc.) and doesn't seem to cause problems there.

2:05pm User3 says "I don't know where I'm at." upon opening a new ruleset editor. He then says, "what if I want to go to functions?"

User3 is a bit confused and is not sure how to proceed. This is understandable in part because he has been given little guidance.

2:05pm User4 wants "to go" to a Utility Group and says so. He tries "Add Utility Group" which creates one. He then double clicks on newly created line and opens the group editor for this utility group.

2:10pm User3 wants to start by entering low-level functions first. He has opened a group editor (I think by referring to User4) and has renamed the group by editing the "Name" field in the editor. Now he wants to "find slots", since the first function he wants to create is just a slot reference. He tries to type directly into empty group editor list (which is supposed to contain lines for the functions which this group contains) to no avail.

2:10pm BUG - User4 changed name of function using "Name" field in function editor. This changed the name in the function's group editor but did not change the function editor's title bar. The user didn't notice this.

12:10pm User3 (with help) has created an empty function and opened it in an editor. Now he wants to type the function body in the text field that is to hold the function's arguments.

This is a valid problem. In the first place, the layout of the function editor's name, argument, type and body are not placed in the expected order from a programmer's perspective. They are placed in: ***name_text_field*** *(user editable)* ***"("arguments_text_field*** *(user editable)* ***")" function_type_option_menu body*** *(structure editable)*. If anything the order ought to be as in most languages ***type name args body***. In addition, although the arguments text field in surrounded by parentheses there is not editing support given the user in entering these. Although there is parsing of the arguments when the user hit return.

Other than the layout, it is unclear how much labeling would be warranted. In the first place, a label for each of these would certainly help the novice but might annoy the more experienced. On the other hand, the whole point of the structure editor is to help the novice, so this might be a good approach.

One point of note here is that labeling would likely help novice users and be ignored by more experienced users. And since neither user type would have to actually type in these labels, any

inconvenience would be fairly minor. In a purely textual language, any labels added to facilitate novice learning would have to typed in by the user. This is extra typing would become potentially cumbersome to frequent users of the language. This is akin to what COBOL programmers must deal with (NOTE: Can I find any literature to support that claim that this extra typing is a hindrance to programmers?).

2:15pm Neither user seemed to know how to get started on creating a function using the structured editing supported. I let them look around the menus a bit to see if they could see which option would help them. Since neither of them found and recognized the palette as the correct choice within a minute or two, I told them about it. I did not tell them how to use it, but let them play with it a bit to see how it worked. They seemed to get the basic idea.

I'm not sure if the fact that they didn't recognize the palette as a means of creating functions is really a problem. To a degree, I would like to have everything obvious from the outset, although I suspect this is not always possible. Perhaps instead of "Palette", "Editing Palette" would be better since it would tell the user that this option supported editing. If so, this is another example of using labeling to help the user.

There seems to be a principle here that I have, to a degree, neglected. I have often overlooked using terminology which is familiar to the targeted user and instead used terminology which is concise and/or more attuned to a programmer. In addition, I have sometimes opted for context dependent conciseness (as in the lack of labels in the function editor) at the expense of clarity.

2:15pm User3 found the slot selector and successfully used it to place a slot in the function body.

This is a minor victory, in that User3 was able to create this simple function with relative ease once the existence of the palette was pointed out.

2:15pm User4 asked how to get rid of part of a function. He had added an IF statement and wanted to get rid of the antecedent which he had created. I told him to look at the options under the "Edit" menu. He then had some trouble selecting the part of the IF statement he want to clear. He finally got it and cleared it.

There are a couple of points to consider here. The first is that the use of the "Edit" menu is common in most applications that the users would have used. Yet, this is not common in *RiverWare* (although it *should* be). Still, this is a good approach and one that I do not feel needs change.

One feature of the "Edit" menu, which I feel is good, is that is contains standard options ("Cut", "Copy", "Paste", "Clear") which are only activated when the selected expression allows this operation. Thus, at any given time, some or all of the options may be grayed out.

The second point is more problematic, although it was not unexpected. This is the method used to select parts of the function and rule. Basically, each function and rule is made up of an internal expression tree. This expression tree is represented visually. In order to select some part of this tree, the user needs to place the cursor over the expression and click the mouse. This will cause the expression to be selected and highlighted to show this. This is convenient when selecting a leaf of the tree.

It is less clear, however, how the user should select a non-terminal expression (e.g., "a + b"). In these cases the user needs to select the operator (e.g., "+"). This holds true for complex expressions, such as the IF expression. In these cases, the user may select any part of the text/image which makes up the complex expression but which is not part of this expressions sub-expressions. For example, the user could select the "IF (", ") THEN", "ELSE", "ENDIF" and

have the entire IF expression selected. If the user selected the antecedent or either of the consequents, however, only these sub-expressions would be selected.

I am not entirely happy with this approach for selecting parts of the expression tree. Although it works and allows the user to select any part of the expression tree, it is not necessarily intuitive. I'm not sure, however, if there is a better approach. One option I had considered and may still do, it to allow the user to click and drag a rectangle and to select the portion of the sub-tree that encompassed the selected sub-expressions. I like this option, although I am not convinced it would be any more intuitive. In addition, it is difficult to implement. Either way, this is a case where a good tutorial/user manual would be warranted.

2:15pm User3 was a little confused when he wanted to save the ruleset from the function editor. The menu title is "File" and the menu item title is "Save". The user thought that this meant that he was saving only the function to the file and that to save the entire ruleset would be a different action.

This is labeling problem again. Instead of "Save", "Save Ruleset" would be clearer. This same labeling applies to all editor types. If loading and saving partial ruleset is supported, it might be necessary to allow for saving groups, rules and functions separately. In this case, new menu items ("Save Function" and "Save Function As") would be required.

As it is now, there is some ambiguity regarding the "File" menu. Some of the options apply to the thing being edited. For example, "Close" applies to the editor and not the entire ruleset. On the other hand, "New", "Open...", "Save" and "Save As..." apply to the entire ruleset. "Print" is not currently implemented, but it would seem best to have it apply to the thing being edited, rather than the ruleset.

I will need to do some thinking in order to determine what the best way to handle this ambiguity. One solution is to replace "File" with "Ruleset" and place all ruleset specific operations here. "Close", which is traditionally placed in the left most menu could not go there since it applies to the editor and this could be confusing. Another solution is to more clearly label each option under file. For example "Save Ruleset", "Close Editor", etc. This would be better, but would not address the use of "File" here, since only "Save" and "Save As" deal with a file. Using "Editor" would also not work, even though it is closer to the truth, since it is too similar to "Edit".

2:20pm User3 wanted to create a second function. At first he tried to do so from the function editor he currently had open. He quickly realized that this was not possible and went to the group editor and created the new function.

I'm not sure there is anything in particular wrong here. The user had an initial misconception which he quickly rectified. On the other hand, perhaps having the "New" menu item could apply to functions and would create a new function and open it in the function editor. It would be placed in the original functions group by default. While seems like a reasonable idea, it would depend on making the menu consistent with this operation and that of creating a new ruleset. And, as with the idea of saving portions of the ruleset individually, it would depend on whether there were enough operations for each editor to warrant its own menu. (NOTE: Do I have anything deeper to say about the organization/naming of menu/menu items?)

2:20pm User4 asked, "If I create a new function, does the old one stay up?"

This is a reasonable question. Some programs, such as Galaxy's vre (Visual Resource Editor) only allows one editor in some circumstances, although this is not consistent. I'm not sure there is anything I can do about this short of good initial documentation and letting the user try and

see whether this is true.

2:20pm User3 said, “It seems like I need to re-open the palette each time I need to use it since the buttons are grayed out.” He then tried to close and re-open the palette and saw that this comment was not true.

This happened in the context in which the user was trying to edit using the palette but where he was not aware of how the current editor selection affected the palette buttons activeness. He seemed to think that just clicking within the function editor was enough to activate the palette buttons.

This misunderstanding is based, in part, on the policy that only the palette buttons which correspond to valid substitutions are enabled. Since the user had not selected any expression nothing would be enabled. I’m not entirely sure what the user thought he could do with the palette buttons if they were enabled and nothing was selected. Perhaps he thought that the palette was not used as a substitution mechanism but an insertion mechanism. I suppose this latter perspective is understandable, although I don’t think it would work very well in a structure editing environment.

One option is to not simply grey out the palette buttons, which is probably a behavior expected of experienced users, but to also put some kind of explicit indication of the fact that none of the palette buttons are enabled because nothing is selected or the thing selected is of some invalid type (ERROR) and has no valid substitution. Ideally, either of these would have to be somewhat intrusive so that the user could not miss them if s/he were looking at the palette. A more practical approach would be to include a message area in which the type of the current selection or “no selection” was printed.

As always, good documentation would help, although I am hesitant to always fall back on this. Good documentation should be used to help a user understand a good tool, but should not be used to prop up a poor tool.

2:25pm This is a continuation of the previous comment... I told User3 to leave the palette open and to select an expression, which would in turn activate some or all of the palette buttons. He did this and proceeded, on this own, to convert <expr> to <expr> - <expr> and then use the slot selector to substitute two slots for the empty expressions.

BUG - He did, however, double click on the slot selector and get two copies of the slot selector. This is a fairly innocuous bug.

The above example shows that the use of palette, once very briefly explained, seems to be intuitive. Perhaps this could be used to argue that a few dialog-based hints can go a long way in helping the user use the tool without relying on documentation.

2:25pm User4 could not read the entire name in the slot selector and requested a vertical scroll bar.

This is not really related to my stuff, since I am just using *RiverWare’s* slot selector. The slot selector was developed without considering how long the names of objects and slots might actually be. In fact, the names are sometimes extremely long.

2:25pm User3 tried to single click on the name of a function listed in a group editor. A double click is necessary here. He mentioned that it seemed unclear when a double click and when a single click was necessary.

In order to comment on this, I need to catalogue where single and double clicks are permitted

and what actions result for each of the editors.
Ruleset/Group Editor
- single click:
  treeview indicator - display group's contents, unless its group editor is already open
  group, rule, function name - open a text field for in-line editing
  group, rule, function check mark - toggle active state
- double click:
  group, rule, function line (not treeview/name/check mark) - open an editor for selected item

Rule/Function Editor
- single click:
  expression - select and highlight expression and enable palette buttons
- double click:
  terminal expression - open a text field for in-line editing
  user-defined function name - open function editor for that function

Based on the above, there seems to be one case in which there is an inconsistency with respect to how single and double clicking works. In all but one case, a double click invokes some kind of editing. This exception is when the user clicks on the name of a group/rule/function from within a ruleset or group editor. A double click might be more consistent here, but it also might conflict with the use of double clicking the line to open an editor for the clicked line. On the other hand, as things stand now, if the user double clicks within the name, Galaxy interprets this as two single clicks and will open the name for editing.

2:35pm User3 selected the binary expression object.slot[] - <expr> and then used the slot selector to select a slot. This replaced the entire binary expression with the selected slot. He intended to replace only the empty expression with this new slot. He said he thought this happened because the slot selector was open, although I don't really understand what he meant by this. At any rate, he tried this again and saw that the reason it happened was because the entire binary expression was selected.

This behavior is intended although I suspect there might be an argument that a user should not be able to make such a destructive substitution. A warning here might be warranted. Although it might be intrusive, it could allow the user to suppress future warnings of this type. It seems that even if it is intrusive, it might be nice since it will prohibit the user from inadvertently destroying a potentially large part of an expression tree.

It should also be noted that binary substitutions are not destructive in that they place the selected expression in the LHS of the binary expression. The fact that some substitutions are destructive and others are not seems like a reasonable argument for warning the user when they are about to destroy some part of their expression.

An alternative which I don't think is all that good is to not allow destructive substitutions at all and to require the user to do an explicit clear on an existing tree before they are allowed to make a substitution. Thus destructive substitutions could only be performed on empty expressions. While this would be safer, I suspect that this would also make editing more cumbersome than necessary.

2:35pm User3 selected the left side of <e1> - <e2> and used "Edit->Cut". This removed <e1> and left only <e2>. While this is correct behavior, it was not what the user intended. The user really intended to clear the expression in <e1>.

I don't really see a problem with this since this is the standard way that cut and clear work. The main problem, however, centers around how the user can get back to where he wanted to be. Since he wanted to get to <expr> - <e2> and instead got to <e2>, he would have to select <e2>

and hit the <e>-<e> on the palette. Yet, this would result in <e2> - <expr>. The alternate is not an option on the palette. As result he would have to select <e2> and clear it, then select <expr> and paste in <e2>. A bit cumbersome, but it works. An undo feature would be better.

2:35pm User4 realized that it was possible to rename functions in-line using the group editor.

This ability and the fact that he found it by chance is, in part, based on the fact that only a single click is necessary to invoke the in-line editor. If, as was discussed above, this was changed to require a double click, it is unlikely that this would be found as easily, although given that a double click is one way to open a rule or function editor, it is certainly conceivable that this feature would be found.

One point of note here is the way in which the location of the single or double click is important. If the user single or double clicks on the name field, the in-line editor will be invoked. If the user single or double clicks on the active check mark, it will be toggle (twice if a double click). If the user double clicks anywhere else, an editor will be invoked.

2:35pm User4 wanted to place a function in the antecedent of an IF/THEN. He wanted to find a list of available functions where he could select one and place it in the antecedent. Although he seemed to want to do this by substitution (as is done using the palette), he didn't look for this list on the palette. I am not entirely sure where he thought he could find this list. I suggested he look at the palette, which he did and subsequently found the function he was looking for in the palette's list of available functions.

I think this was just an oversight on the user's part. I think placing a list of available functions on the palette is consistent with how the palette is to be used. I guess the only comment I would have is that it might be nice to separate the list of functions into pre-defined and user defined functions.

2:45pm User3 asked how he could add a units designator to the end of a function call. I told him that this was only necessary for literal numbers.

This question was unexpected although it is understandable, I suppose, in that a function returns a value and this value needs to have units. From an orthogonality point of view, assigning units to a function call makes sense. Yet, the intent of the language is to have the function return the value in the units desired. In this case, the user had defined a function which was the difference of two slot values. Since each of these slot values have units, the result will also have units which will be assigned to the function's result.

This confusion might well have stemmed from the fact that this language does not require the user to specify units for most equations. The exception is when a literal is specified. In all other cases, a value (whether from a slot value or a function return) has units associated with it and these units are used in all arithmetic expressions. The result of these arithmetic expression also has units, which may or may not be the same as the expression's operands, depending on the operation. If the units are compatible for a given operation, the evaluation of the expression will usually succeed (an exception is division by 0) and generate units for the result.

2:45pm BUG - User3 found a bug which allowed him to place a NUMERIC function into the antecedent of an IF expression. Only a BOOLEAN expression is allowed here. Then when the user wanted to replace this function with a BOOLEAN binary expression (i.e., IF (func1()) to IF (func1() > func2())), he was not allowed since he would be replacing a NUMERIC expression with a BOOLEAN expression.

This second restriction is, generally speaking, valid. The exception is when the user is allowed

to add something illegal which gets them into a bad state.

2:45pm User4 was unsure of the purpose of the return type in the function editor. I told him.

There are a couple issues here. The first is the layout of the function editor which was discussed above. Fixing this would certainly help in removing most of the confusion about how the return type should be used. On the other hand, this user may not have known what to do with the return type even if it had been clearly identified. Since a function needs this, it is probably a training issue to some degree.

It does not seem like an acceptable option to remove the return type from the function editor. For although it is possible to determine the return type from the function's body, this means that a function's body must be defined before the function can be used. Based on some pre-user testing I determined that this was too cumbersome since it required the user to, at a minimum, put some rudimentary place holder in all functions in order to use them. Not only is this cumbersome, it is also potentially dangerous, since it would mean these functions would be syntactically valid when the user might have wanted to keep them invalid to insure that they are not inadvertently used.

2:45pm User4 double clicked on a terminal expression and entered a literal using in-line editing without any prompting from me. He lost these changes, though, since he clicked outside of the text field instead of hitting a carriage return within the in-line editor. He then correctly entered the desired expression.

It is good to see the user experiment and be able to edit successfully. As for losing his changes, this was mildly annoying although it was easy enough to correct in this case. Although one might argue that it would be better to accept changes whenever the user leaves the text field, this would make it awkward to support the user who wanted to cancel his in-line editing session without changing the original expression. While there are options (menu, etc.) that could work, none seem better than the method implemented.

2:45pm BUG - User4 typed a word (maxOso) using the in-line editor into the second consequent of an IF expression. He got a parse error, which is correct, since this is interpreted as an unbound variable, yet the expression he was trying to replace was not restored. Instead, the word he typed in remained.

The main point of interest here, I think, is the error message. While it might be true that the interpreter sees "maxOso" as an unbound variable and thus invalid in this location, telling the user this is not necessarily helpful. Nor is the message I display which is that the entered text is not valid at this location. It is likely that the user meant to type in a function call, such as "maxOso()". While it is not feasible to know what the user really intended for any given illegal action, it seems reasonable to provide a more complete error. For example, "The text entered "..." cannot be placed here. A NUMERIC expression is required and this expression does not have a type." It is difficult to say if this is any better.

2:45pm User4 had <e> + <e> > <e> as the antecedent of an IF statement. He asked if the precedence was as expected. I told him it was.

This is a valid question/concern. In this case, it wouldn't really make sense if the precedence were not as expected, since this would result in a number plus a boolean, which is illegal. Yet, given that C allows mixing numbers and results of boolean operations, this is a reasonable thing to ask.

See the next entry for a continuation.

2:45pm BUG - User3 had entered <e> - <e> - <e>. I noticed that this expression is potentially ambiguous, although he didn't comment about it.

The reason this is a problem is the interpretation of this expression is based on the underlying tree representation and not necessarily interpreted left to right. Thus, without parentheses, it is not obvious which minus will be executed first. This is a class of bugs which I fixed, but I seemed to have missed this one.

2:45pm An observation about the way User3 and User4 program. User4 programs using a top-down approach, while User3 uses a bottom-up approach. User4's approach is closer to how the structure editor requires the user to work. For example, if the user needs to enter a boolean expression which is comparing two numbers, he must enter the boolean expression with place holders first and then enter the numbers. He cannot enter a number, an operator and another number. This restriction is intended to help the user construct only valid expressions. What I find somewhat interesting is that User3, even though he thinks in a bottom-up approach and occasionally has to back up and code a different way, seems to get the language and environment much more so than User4. I suspect this is a reflection on the user. It would be interesting to see how well User3 would do if he was top-down and how poorly User4 would do if he was bottom-up.

2:45pm User3 comments, "This isn't too bad. This is pretty intuitive."

3:00pm User4 tried to copy a function name from within the group editor by highlighting it in the in-line editor. This is not supported.

This seems like a reasonable thing to do, although it is unclear what the user would do with the copied name, since the environment does not support copying things which are not objects in the language (i.e., expressions, rules, etc.). I suppose I could allow the user to copy text and paste it into another in-line editor. Yet this is supported by most window managers by default and would require the user to deal with copying/pasting things which are not compatible.

3:00pm User3 wants to create a rule and realized he needed a policy group to place the rule in. He was able to do this, yet said, "I want to go to a policy group."

I am not exactly sure what he meant by this. Since he was able to perform the action successfully, I assume he meant "create a policy group", although I can't be sure.

3:00pm User3 wanted to create a rule with three rule statements. He said he didn't know how, yet when he saw "Rule->Add Assignment...", he asked if this was what he needed. I let him try, which worked. Then he was able to use the slot selector to place a slot on the LHS of the new assignment statement.

3:00pm User4 created an assignment statement without prompting, although he may have overheard my conversation with User3.

3:00pm User3 created second and third rule statements.

3:00pm User4 asked, "What am I solving for?".

He wanted to know what should go on the LHS of the assignment statement.

3:10pm User3 commented. "You don't have an OR command expression." I asked why he thought is was a command expression and he answered, "because it is not a math expression and I want to do a command." Within about 20 seconds, he found it. I pursued this a bit in order to

determine where the misunderstanding was. He said that “Binary Expression” on the palette was too computer science specific but in hindsight felt it was all right given that he could not think of anything better.

This seems to be another case where labeling is important. First a little background. On the palette, buttons are grouped and boxed. “Command Expressions” is the label given to the IF/ FOR/WHILE/ELSE expressions and “Binary Expressions” is the label given to all binary expressions, regardless of type.

3:15pm User3 wanted to convert “<o.s> - IF...” to “<o.s> - function ()”. In order to do this, he had to create a function, copy and paste the IF expression and place the function call in the statement. He did this successfully.

This is a good example of some fairly complex editing which the user performed well.

3:15pm User3 is confused regarding the distinction between cut and clear for binary expressions.

Cut will remove the selected operand for the binary expression leaving the other operand. Clear will make the selected operand an empty expression. I’m not sure there is much to do here other than document the behavior, since it seems both useful and consistent with other interfaces.

3:15pm User3 successfully added an ELSE to an IF expression.

This requires selecting the IF’s consequent and then pressing the ELSE button on the palette. This is encouraging as I thought this might not be intuitive.

3:15pm User4 finished creating his ruleset and wanted to try it out. He tried to load it using the “Load” button on the rule editor and got an error. He then tried to load using “Load” button on ruleset editor and again got the same error. He then tried to load using the “Load” button on the group editor and again got the same error. This time, he read the error message: “Rule has an invalid rule statement” and thought this meant that it didn’t like the names he had used.

Perhaps it is confusing to have “Load” buttons on each editor. I did this to enable the user to load/unload at anytime without having to find a particular editor (ruleset is likely candidate). It really didn’t occur to me that a user would think that each button might generate different results. I suspect this is a labeling issue as well. The labels are “Not Loaded” and “Loaded”, reflecting the current state. Pressing the button changes both the state and the label. Perhaps a more verbose “Ruleset Not Loaded, Press to Load” and “Ruleset Loaded, Press to Unload” would be better, although I am hesitant to make such a long label. Really, though, this wasn’t his issue, so perhaps “Ruleset Not Loaded” and “Ruleset Loaded” would suffice. Since each editor would have these labels, it would presumably be clear that the action pertained to the ruleset and not the particular part of the ruleset represented by the editor.

3:15pm BUG - User3 keeps having expressions disappear.

3:20pm User3 uses “Group->Check Validity” to check the validity of a group. Group is valid, so he tries to execute the simulation and gets the error: “No ruleset loaded”. He returns to the group editor and successfully loads the ruleset. He then runs the simulation completely, although because the user only created a partial ruleset, the results are also incomplete.

This is a success, since he was able to construct, validate and use the ruleset.

3:25pm User4 still needs to pinpoint the problem. Since the error messages are not as good as they could be, I help him find the error. He corrects it and successfully loads the ruleset.

The users are finished, so I ask if there are any questions.

User3 asks, “How would you copy a function from one ruleset to another?” I told him to give a try. He did it on his own, although he didn’t appear to realize what he had done. Then, after realizing what he had done, he said, “That was very intuitive.”

The operation is fairly straightforward, although the feedback is not so good. If the group to which a rule or function is to copied is empty, a small triangle, indicating that the group has members, is added on the left most side of the group line. This may be missed by the user. If the group already has members and is closed, there is no feedback. While users would likely get used to this, it might be good to have some more obvious feedback. It is difficult to say whether this would prove annoying to a more experienced user.

User4 asked what the difference between copy and duplicate was on the ruleset editor. I asked him try it out. He did so and understood.

At this time, User3 and User4 were finished and were ready to switch roles with User1 and User2. Overall, I felt that User3 and User4 had a positive impression of the system. Before User1 and User2 started to use the system, the users took a short break.

F.2.3. User1 and User2

As with the first two users, User1 and User2 were also going to write a portion of the ruleset which we had discussed before lunch. I gave them the same instructions as the first group and sat between them while they worked.

4:05pm User1 added a policy group and utility group to start. He then added a rule to the policy group.

I think he was doing some experimentation, since he really didn’t need to create the utility group. He seemed to know what he was doing since he created the rule correctly.

4:05pm User2 did not appear to know where to start once he had opened the ruleset editor. While looking around, he opened the sub-window and added a description. He switched to rule priority. He created a policy group and a rule in it. He opened the rule. He then opened the palette and studied it for about a minute.

It appeared that he really didn’t have a good sense of what he wanted to do or perhaps how he should do what he wanted. I say the former in part based on my observations and in part based on the fact that he did other things, such as add a description, rather than focus solely on creating the group and rule. I guess he figured things out fairly quickly and seemed to not have too much trouble doing so.

4:10pm User1 asked, “Do I use Add Assignment?” with the rule editor open. I answer yes. He created an assignment statement. While trying to create a function, he double clicked on the LHS of the assignment statement while trying to create a function. He then looked at the options under the Rule editor’s Rule menu. He finally found the create function option on the Ruleset editor, yet he didn’t realize he had created the function. I suggested he look at the Group editor where he found the newly created function.

One thing to consider is that the user must create a function before he can use it in expression. Thus, the user’s attempt to create/use a function from the rule editor cannot work. This is a limitation and forces the user to do some more top down development. I don’t think this is too limiting, since if the user wants to use a new function in an expression, he presumably knows

what he wants to name it and it isn't much effort to create it from either the ruleset or group editor. There may be some other options, such as creating or asking if creation is wanted when a new function is added to an expression using in-line editing, although this might be more than is really necessary.

The other issue about not knowing that he had created the function has been discussed previously. I guess the main issue here is that the user will know that a function has been created from now on and even though this is a bit confusing the first time they do this, it seems that any explicit notification would prove overly cumbersome in the future.

4:10pm BUG - I noticed that the group editor's title does not identify whether or not this group is a policy group or utility group. This is a fairly minor bug.

4:10pm User1 tried to type the entire contents of a function into a function editor using the in-line editor. The in-line editor is fairly small and would not be a good way to type anything but a very simple function body. I told him that a better way existed and he should look for it. He first tried opening the sub-window, found palette and put in the IF expression.

What this user tried to do is understandable given that in-line editor is allowed. The user could have successfully typed in the entire contents of the function, but it would have been very difficult since the in-line editing text field is much smaller the text that would have to be typed in. I'm not sure there is anyway to prohibit this if in-line editing is allowed, which I think it should be. It seems the relatively small text field should serve as a reasonable deterrent, although it is obviously not fool proof. I suspect this is a trade-off that is worthwhile.

4:15pm BUG - I noticed a bug in which the numeric binary buttons are enabled when the IF expression's antecedent is selected. This has been noted above and must be fixed.

4:15pm User1 tried to find slots in the palette's function list and was not able to (since they aren't there). He looked around a bit longer and found slot selector which he opened and used.

4:15pm User2 was looking at the function editor and trying to determine how to define a function body. He said, "Parentheses on a function indicate that some kind of expression belongs there."

Here again, the lack of labeling on the function editor has caused confusion.

4:20pm BUG - User1 added something to the arguments area of the function editor, hit carriage return, got an "Invalid" notice and then a core dump.

Labeling confusion. The notice correctly identified that a problem existed, although it would be helpful to also tell the user what is expected rather than making them try until they get it right. Bug must be fixed.

4:20pm User2 tried to place an assignment statement in the body of a function by placing a <e> == <e> in the body. This is legal and would result in a function which returned a boolean. It was not his intent.

Perhaps "==" should be replaced with "EQ" or something like that to make it more obvious that assignment was not the intent. I suppose even this could be misinterpreted, although probably not as often.

There is perhaps a larger problem here in that an assignment statement is not allowed in a function, so even the user's intent was not correct. I suspect he tried the "==" because this was the closest he could come to what looked like an assignment. This may be something which

needs to be taught since a function in this language is different from what these users are probably used to.

4:30pm User1 asked, “Can I create another function while still editing this one?”

Ran into this one in the previous test. I’m not sure if there is anything to do about this other than document it.

4:30pm User2 asked, “If I close a window [an editor], will I lose my changes to the contents?”

The answer to this one is no, since the ruleset is maintained as a whole and saves are only relevant on the ruleset level. Here again, I suspect that documentation would be warranted, since it is not obvious which answer is correct based on the editors themselves. One change which *might* clear things up would be to change “Save” to “Save Ruleset” since this could imply that only the ruleset could be saved. Documentation is better.

4:30pm User1 was able to quickly and correctly use the slot selector to add to a function an expression of the form: object.slot[] - object.slot[].

This is encouraging.

4:30pm BUG - User2 was trying to create a function of the form obj.slot[] - obj.slot[]. He first tried to double click to type in slot name by hand. Then, without closing the in-line editor, he opened the slot selector and selected a slot. But since the in-line editor was open, he didn’t notice that he had selected a slot. Then when he went to close the in-line editor, it parsed what he had written and overwrote the slot he had added using the slot selector.

I should not allow both in-line and palette to be used at same time. Perhaps, I can disable latter when in-line editing is enabled.

4:35pm User1 brought the *RiverWare* main screen to the foreground. In doing so, he covered up all his 10 or so editors. He assumed he had lost them and went to start over. I told him to look behind the *RiverWare* screen.

This is more of a window manager issue, since it is possible for a user to lose his windows and I can do little to prohibit this. I do one thing, though, which should help. If a user has a *named* ruleset open and tries to open it again, I will just bring it to the front. This does not help in the user’s situation, since his ruleset had no name and I want the user to be able to create more than one unnamed ruleset at a time.

4:45pm BUG - User2 saved his ruleset file to a non-readable directory. He received no warning and the save didn’t happened, so when he closed his ruleset, he lost all his edits.

This is a significant bug which has been fixed.

4:45pm User1 said, “That was easy” after creating five functions.

4:45pm BUG - User2 opened two slot selectors.

This is not serious, but should not be allowed.

4:45pm User1 paused when he had to enter “0 [cfs]”. He didn’t know how he could enter this since there is no support for this on the palette. He tried to type on top of a selected expression, which didn’t work. He then tried (<E>) on palette which is used to add/remove parentheses to the

selected expression. I told him that he couldn't do this using the palette. He found "Rule->Edit Element" which worked.

User could have also used double click to invoke an in-line editor. I suspect that it would be better to put a text field on the palette which can be used to fill in any terminal expression in much the same way that in-line editing works. This text field could be activated when the user selected a terminal and in-line editable expression.

4:55pm BUG - Rule name title doesn't change when the name of the rule is changed.

5:00pm BUG - User2 entered "IF (a+b)...".

This has been noted above and is illegal.

5:00pm User2 tried to place a rule in the rule "Execute Only" section, rather than creating a rule statement. He realized this was incorrect and after looking at the palette, he finally found and added an assignment statement.

A couple of things here. The execute only section is pretty clearly labeled, so I think this was an oversight that should not be changed. As for adding a rule statement, I opted to leave them off the palette since I felt the palette should be exclusively for expressions which could be used by any editor and to have editor-specific statements be added from those editors. This may be a bit confusing at first, although I suspect there would be an equal, if not greater confusion by placing the statements on the palette. Recall the above attempt to have a function do an assignment.

5:05pm User2 selected the LHS and added "maxOso" in an attempt to create a local variable. A parse error is generated.

I need to generate clearer errors when a user adds some thing incorrect. The more context sensitive the better. In this case, I could state that I am expecting an object.slot[] expression, since that is all that is valid on the LHS of an assignment expression.

5:15pm BUG - User1 is losing expressions. This is serious.

5:20pm BUG - User1 says that the slot selector sometimes stops working.

5:25pm BUG - I notice that rules and functions can be created out of place and that functions in the palette's function list can sometimes be out of order. The user's don't seem to notice and neither of these bugs is very detrimental.

5:35pm User2 entered "0 [cfs]" using double click generated in-line editor, although he first tried to find some way of doing this using the palette.

Again, I might want to allow for in-line editing via the palette.

5:35pm User1 feels that validation error messages are too cryptic.

He is correct here. In fact, all error messages need to be far clearer. They need to identify exactly where and what the problem is.

5:35pm BUG - I notice that the slot selector and palette remain up even after the last ruleset editor is closed. They should be closed at this point, although they do no particular harm staying open.

5:40pm BUG - User1 can't load a valid ruleset. This is serious.

5:50pm BUG - User2 can copy and paste a rule statement into a function. This is serious.

6:10pm User2 wants to add an ELSE to an IF expression but can't get ELSE button to become enabled because he has selected only a portion of the IF expression's consequent.

This is a tough one in that it is doing what I intended and what seems reasonable, yet the user is having trouble here. I can imagine giving the user the ability to add an ELSE to an IF expression by selecting part of the consequent if the consequent is fairly simple, but since the consequent can be arbitrarily complex (within the scope of the language), this might lead to ambiguity in some expressions. Even though there are undoubtedly ways around this, I think it is best to keep things as is and requiring the user to select the entire consequent to add (or remove) an ELSE. Documentation would help here.

6:10pm User2 would like to select expressions using click and drag.

I had originally envisioned this as the method of selecting expressions. Because of the need to develop quickly, I used the single click method. I would agree that click and drag would be a good addition and hope to add it to the system eventually.

6:10pm BUG - User2 pasted an IF expression onto the LHS of an assignment statement.

This a serious bug. It is unclear what the user was trying to accomplish here. Either it was a mistake or the user is clearly confused about what is syntactically correct. Either way, this kind of thing should be strictly prohibited.

6:20pm BUG - User2 deleted a rule statement and the following rule statement's formatting was corrupted.

This is a moderately serious bug, which can be cleared up by opening and closing the editor. Until then, editing is possible, but not wise.

6:30pm User2 feels that the function list on palette is difficult to use since it is regenerated each time the selection is changed and he must scroll down to the bottom each time he needs to get to a user-defined function.

This is good observation and something that should be fixed. First, though, it should be stated that updating the function list as the selection changes is necessary, since each selection may have a different list of functions. There are a couple of options to make this more bearable. One is to keep track of the items that are in view and try to bring them back in to view when the selection (and thus function list) is changed. This could be difficult since these functions may not be present for all selections. Another is to check if the list actually changes and only refresh it if it has. This latter idea is doable. Lastly, splitting the user and system defined functions would make it easier to find functions and alleviate some of the frustration.

6:40pm BUG - User2 feels that selecting parts of an expression of the form a-b-c is not intuitive since selecting one "-" get a-b and selecting the other "-" get a-b-c.

In part this is a result of the tree structure. Parentheses are necessary here since this expression is ambiguous. In addition, It might be possible to allow selection of either a-b or b-c if binary expressions are allowed to take more than two operands. For example, a-b-c would become a tree node with three children whose operator is minus. This would have to assume left-to-right precedence and if the user wanted otherwise, he would have to use parentheses. While this

would be convenient it would be a good deal of work and, for now, it is unlikely to become a high priority item.

My overall feedback on this second session was that it did not go as well from a user satisfaction point of view as the first session. It was good in that it identified many bugs and features that need to be looked at and corrected, yet I think the users started to get a bit frustrated. In addition to the bugs which I think caused some frustration, the session took an hour longer than the first (in part because of the bugs) and it went well past their normal work day's end. As such, I think the users were getting tired.

The bugs identified need to be fixed (they have been) before the next user testing. One note is that this user testing was done at a time when we knew we would find a lot of bugs. The users were informed of this and told that they were the first group to really use the system. I think they understood.

F.2.4. Day Three

On Thursday (10/9/97), All four users worked together to create their entire ruleset. They decided to split the coding responsibilities between them. They had already designed the rules and functions and just needed to enter them. Once each had entered their parts, they would bring them all together into one ruleset.

Working together was not something I had explicitly designed the language and editors for, so when they told me this, I was unsure how it would work out.

11:05am User3 wanted to convert a volume to a flow. I had told him that a predefined function existed to do this. He found the function, yet tried to use the function as a conversion factor rather than a function. That is, he tried to enter the following: volumeValue * VolumeToFlow (?,?) and didn't know what do with the arguments. The proper way to use this function is ***VolumeToFlow (volumeValue, date)***.

This fundamental misunderstanding might need to be addressed with documentation, since use of functions in the manner is quite common. It seems that what the user was trying to do was to multiply the volumeValue by a conversion factor which would result in the value as a flow. As such, one might think that including some pre-defined conversion factors which could be used in situations like this. Unfortunately, this will not work, since these conversions must be made in the context of a date value. This is because some of the units that are used are monthly units (e.g., acre-feet/month) and their value changes based on the number of days in the month.

11:10pm User3 wanted to move all functions from a utility group to a policy group. He wasn't sure if copy/paste would work or if paste would overwrite the selected functions in policy group. Since it is impossible to deselect a line (function, rule, group) in either the group or ruleset editor, the user wasn't sure what to do since there was always the possibility of overwriting something.

This is something that should be changed, since paste generally means to overwrite the selection. "Append" is a better term for this option and would probably allay a lot of confusion. As for the line selection issue, a line always needs to selected when an append is to be done, since I must know into which group I am appending.

Also, it turned out that copy/paste would not work since he was trying to copy from a utility group in a ruleset to a policy group in the same ruleset. This would have resulted in a duplicate function name which would be caught and prohibited. The user would have had to do a cut/paste.

As a aside, cut should probably be “delete” since that goes better with append.

11:10am User1 and User4 were unsure what the two place holders were for a table slot.

Documentation and a better understanding of *RiverWare* would help. It might also be nice to have some descriptive, context dependent place holders. For example, instead of TableSlot [<numeric>, <numeric>], I could use TableSlot [<row>, <column>]. While this helps in describing what is expected, it doesn’t help with respect to what type is expected. Perhaps TableSlot [<row:numeric>, <column:number>] or use of color corresponding to type would be helpful. While I like the idea of providing more information, I am hesitant to include so much information that the expressions get long and difficult to read.

I think part of the confusion was related to the fact that they are used seeing SeriesSlot [], which takes a date/time.

11:15am BUG - User1 is losing expressions again. Seems related to copy/paste.

11:20am User3 is unclear about how to create a complex boolean expression to use as an antecedent of an IF expression. The boolean expression is: MBD > TTD OR TTD - BLDC <= SJD[] AND TTD - BLDC <= MLOD. The user wanted to create this expression by typing from left-right when in this environment, it must be created top-down.

Unless the user wants to do in-line editing, which would be difficult and error prone for an expression of this size and complexity, top-down creation is necessary. This is a liability of a syntax directed editor. The only way to get around this requirement would be to turn off the syntax checking so that the user could construct anything. While this might be helpful for some users, it would also take away any gain there is to be had with respect to creating syntactically valid expressions. I suppose I could imagine allowing users to turn off the syntax directed palette restrictions. The resulting code would be verified for correctness at simulation run-time, so this would cause no harm in that respect. Perhaps this would be a good “expert” mode.

11:20am User3 and User4 wanted to do their work separately with one creating the rules and the other creating functions. When finished, they would combine their code. Yet, this really didn’t work out to well since the rule writer needs access to the functions which are being created by the other user. Since the functions don’t exist in the rule writers ruleset, they will not appear in the palette’s list of functions and thus will not be able to be entered.

In order to get around this, the rule writer just entered the function names without “()”s. They thought this would suffice and that once they copied the functions into the rule’s ruleset everything would work. Yet, since the names they had entered were not function calls, they had to replace all these names with the real function calls.

A couple of things. First, these unbound names really don’t seem like they should be allowed. This is especially true if they are used in conjunction with bound values, for example 1+bob. Second, the language supports this type of work separation to some degree. Since a user can create a blank, typed function, the rule writer could create blank functions and use them through the palette. Then when it came time to merge, he could delete all these functions and import the complete ones. This would work without too much hassle. Granted it is more work than if the user had no syntax checking, but then he would lose this benefit.

11:30am User2 wanted to use SumSlotOverTime (objectSlotName, date, date). He tried to use the slot selector to make object.slot[] the first parameter. While this seems reasonable, it will not work in this case, since the slot selector will add a slot value reference (which is a number) rather than a slot pointer.

This is a problem in the language. Currently, the object.slot[] syntax is really of type NUMERIC, since that is the type the slot pointer evaluates to when used in the context of a date or a row/column. There is really no OBJECT_SLOT type in the language although that was the original intent. Both uses of object.slot syntax are needed in the language. This would allow the slot selector to be used in both cases.

As it is now, the user would have to enter a string by hand. This is problematic for two reasons. The first is that the user should be able to use the slot selector since many slot names are quite long and the user should not be expected to type them in, especially when the slot selector can do it for them. The other reason is that if the user types a string, no checking to insure that this string is correct can be done until run-time. Actually, it *could* be done, but this would have to be done on a context dependent basis. Having a true object.slot type would cover both of these problems in an elegant manner.

11:35am User4 wants a slot value for the previous timestep. He tried object.slot[-1], since that was the syntax which was used in the notes that they were given (or wrote as they were designing). Although they had been told, they didn't remember to use @"Previous Timestep".

I suppose this isn't too surprising. The date/time syntax is not all that easy to remember. Originally, I had intended on giving the users some predefined keywords such as CURRENT, PREVIOUS and NEXT to refer to the current, previous and next timesteps, respectively. Time constraints put that effort on the back burner, although I suspect it would be worthwhile to do this now.

11:40am User4 found the line: IF (MBD > TTD OR TTD - BLDC <- SJD[] AND TTD - BLDC <- MLOD) THEN to be so long that is was difficult to read. In fact, the user at first thought that he had lost the right most part of the line since it had scrolled off the edge of the dialog item.

This is a valid observation and one that I had anticipated but didn't have time to fix. Currently there is no way to split a line and the lines can become quite long. The ability to split a line between expressions should be supported.

11:50am BUG - User2 got a core dump when he started *RiverWare* from his home directory instead of from the main executable's directly. Minor, albeit annoying bug most likely related to RulesetEditor.vr file.

11:50am User1 said he could not add an ELSE to the following expression: IF () THEN IF () THEN ... ENDIF ENDIF

I was unable to determine what he was doing here, since I am able to add an ELSE in either of the two valid locations. My guess is that the problem stems from not selecting the entire consequent.

One observation probably made above is how some palette buttons do non-destructive replacements (e.g., binary and ELSE) and other do destructive replacements (IF, FOR, WHILE). There is no way to know which is which without trying them out. This could be confusing.

11:55am User3 wants to change function names and have all calls to it be changed as well.

I had planned to have this as part of the editor, but have not had the time to implement it. This find/replace could be based on expression types or strings and allow for full word versus partial word searches.

F.3. Use Case

On 11/29/97, UserT and UserK were working on a conversion of the particular ruleset for an important model called the 24 Month study. The users were from the Metro Water District (MWD). The MWD is the single largest user of water from the Colorado River. The original rules were written in the original, Tcl-based, language and the users wanted to convert them to the new language. The users spent a day and a half going through the Tcl code in order to refresh their memory regarding what the ruleset actually did. UserK did not really understand the details of the rules. After they felt like they had a good understanding the ruleset, Edie Zagona spent about 1 to 2 hours talking to them about the ruleset and about the new rule language.

After talking to Edie, they decided that they would going ahead and try to do a partial conversion right then. By partial conversion, I mean they would create the rules that they needed and then convert the Tcl functions to External Functions. Thus, they would only create the higher level structures in the new language and leave the bulk of the details in Tcl. They started this process at 2:30 that afternoon and finished with a working ruleset by 5:30 that same afternoon. They were quite excited about this.

This is quite encouraging, although it really doesn't use the language/environment to extent to which I had originally intended and even the users intend on creating the entire ruleset in the new language in the future. An additional positive aspect is that they were able to use the language/environment without trouble.

One interesting point about the use of the language which I did not expect was the use of the External Functions for such large-scale intermediate code. I had expected External Functions to be used for small sections of code which had difficult logic and only in a few well chosen locations. Yet, the idea of using External Functions as intermediate code is a good one, since it would be a very large and difficult job to try and convert the entire ruleset at once. Using intermediate code is nice because it allows the user to slowly convert the code from the new to the old language and to check the answers as they go.

It will interesting to see how this conversion goes. It might be the case that the users will never do a full conversion and that they will simply stop converting at some point and keep some portion of the old rules as External Functions. Of course, I suspect that new ruleset writers will not exhibit this pattern since they are unlikely to have any Tcl-based rules. The four users from the above user test are of this type and do not appear to need/want to use External Functions.

F.4. Second User Test

Wednesday, February 11, 1998. All day.

This user testing was similar to the last one in that it took place in the context of a user training session in which users came with their own problems to solve and spent time studying their problem, seeing how it fit into the new language and then trying the new language/environment.

As before, the user training took 3 days. The morning of the first day was spent discussing the language and how it differed from a traditional procedural language. Since two of the users had used the previous Tcl-based rule language and the other user had used Fortran for quite a while (~20 years), they were familiar with procedural programming. With the exception of one of the users (UserA) who had used the language and programming environment on his own, none of them had had any significant experience with a functional language. The afternoon of the first day and the entire second day were spent discussing their problems and how they would express them in the new language. This latter part took longer than expected.

Some problems with the language came out during this time. Many of these were expected and were parts

of the language which were going to be added when time permitted.

9. Slot names in objectName "." slotName expressions need to be able to be specified dynamically. Now, the language requires slotName to be a constant. Orthogonality. Unanticipated.
10. Local variables are needed for efficiency. Function memoization is another option. Neither seems inherently superior. If local variables are used, it will need to be determined where they are allowed. For instance, they could be allowed as part of the header of a function or in any expression. The former is more restrictive and the latter could lead to unreadable code. Known problem.
11. Function lists need some work. They need to be larger and it would be good to include the entire function prototype in the list rather than just the name. This is easy for predefined functions in that it could be included in the function definition. For user-defined functions, prototype would have to be derived from the argument list. This shouldn't be too difficult. I knew this was a problem, but hadn't had time to fix it.
12. Iterate over dates. The current design for this included a TO operator. startDate TO endDate, using the current timestep. It is clear that a TO/BY operator pair would be needed as well. startDate TO endDate BY timestep. Either of these would return a list of dates. These could be used in FOR/WHILE loops, among other places. This had been designed, but not implemented.
13. List access. Currently, the only places a list is really usable are in a FOR or WHILE loops. The language does not yet support list access. This had been designed, but not implemented.

The three users decided to work together to write a set of rules and functions. This set of rules/functions was only a subset of what they needed to express. In part this was because they wanted to learn the language/environment and in part this was because some of their problems relied on full list functionality. Having this group work together had an interesting result. Unlike the first user test where the users had no rulebased simulation experience, the second group had only one user (UserB) without rulebased simulation experience. One user (UserC) had user the old language/environment for about a year and the last user (UserA, the same as in the pre-User Test) had had extensive experience with the old language/environment and had used the new language/environment on his own for about a month. As such, this group of users was quite a bit more advanced and knew what they wanted both from the language and from *RiverWare*. In addition, since they were working together they were able to help each other and progress more quickly in creating their rules.

In addition, since these users were more advanced, I started them out with an introductory demonstration. I showed them how to create ruleset, groups, rules and functions and how to use the environment. Since I had already gotten feedback on the use of the system without any initial help, I wanted to let the users get a good initial understanding and then see what problems they had.

We started with an introduction of the system. This took about an hour and ten minutes including quite a bit of time for discussion and questions, which are summarized below. For my part, it included a brief overview of the editors and the creation of expressions.

9:50am UserB said he liked the editor/palette combination since it told him what was valid in selected expressions.

10:00am UserC wanted to know if an external function's arguments would be passed in user or standard units. I said that they were in standard units, since the arguments could be changed via arithmetic operations and the user units could be lost. This seemed to be a minor problem, since many of their original Tcl functions relied on the use of user units. The above issue came up as a result of the users' need/desire to use the external function mechanism to slowly transfer their old rules into new rules.

10:10am UserA mentioned that he would like the in-line editor to remain open when he types in something that is incorrect and is given an error. Currently, the user is given the error, the in-line editor is closed and his changes are lost. This is a very good idea and one that I had not

thought of.

10:15am UserA had used the new language/environment before for a about a month. He had requested and been given the system without any training or documentation. He said that he was initially very frustrated when using it, but once he got it figured out has come to like it a lot. This is the user who had spent a great deal of time using the original system and was hesitant to go to the new language/environment.

10:30am UserA mentioned, while discussing the list of functions, that he likes to have the user-defined and pre-defined functions in one list. He does not think that two lists are necessary, although he likes the fact that they are colored differently. This came in response to a comment about the size of the list (too small - I agree here and had planned to change it) and my question as too whether two lists would be preferable.

10:35am UserA said that object . slot expressions do not work as well as they should. The slot needs to be able to be specified on the fly. Now, it is a constant string. An example given is when the object is a data object and contains information for a number of water objects. For instance, there is data object which contains slots named MeadInflow, MohaveInflow and HavauInflow. In order to construct slot names using a reservoir name (e.g., Mead, Mohave, Havasu) as a variable and the flow type as a constant, as in reservoirName CONCAT “Inflow”’, which would then be added to the data object name object, as in: dataObjectName . reservoirName CONCAT “Inflow”’ This seems quite reasonable and is an oversight in the design. This is also a good reason for adhering to the principle of orthogonality.

10:50am UserA needs a way to create a list of dates. As mentioned above, one way is the TO/BY operator pair.

From this point on, the users decided to work together to produce their rules.

11:00am UserC mentioned that they have many “procedures” and that they should split them up and work on them.

11:00am UserC wanted to know how to create dates on the fly. UserA told him how, saying it was very nice and much easier than with the old language.

11:05am UserA mentioned that he really likes having the ruleset be one file rather than multiple files as was required in the previous language/environment.

11:10am UserC wants to use some of his Tcl functions.

11:15am UserA said he would like to have the function name list contain full prototypes.

11:20am UserA again mentioned object . slot, as described above.

11:25am UserA has written three fairly complex rules with lots of IFs (embedded). See RuleSetHeronSJC. He also mentioned that MAX function didn’t keep units, which was a bug that has been fixed.

11:30-12:30 Lunch.

12:30pm The users want to work together in order to create just rules. For these rules, they are using a previously defined Tcl-based rule as a template. They will create functions later.

I will comment on actions/observations that pertain to one individual when necessary. Since

UserB was using the keyboard, I will generally comment about him. As an aside, he was also the least experienced of the group.

12:30pm UserB created a policy group and renamed it. He asked if spaces are allowed in group names.

12:30pm UserB had difficulty opening the group. UserA told him how.

There are three ways in which to open a group. One is to double click on the group symbol to open a group editor. Another is to select the group line and invoke “Ruleset->Open Editor”. The last is to click on the sub-tree triangle. The last one could be confusing here, since the group would be empty and clicking the triangle would have no noticeable result.

12:30pm UserB created a rule and renamed it. He opened the rule and the palette. He then added an assignment statement with UserA’s prompting.

At first UserB wanted to use palette to create the rule. The palette can only be used to create/augment expressions which can be used for both rules and functions. This might be a little confusing. It is consistent in that the palette is applicable to all rule-language editors and any editing that pertains to only one editor type is to be found on the menus. On the other hand, one could argue that putting editor-specific editing commands in the palette in an editor-specific location might be a good idea.

12:35pm UserB double-clicked on the LHS of the assignment expression and got the in-line editor. He wanted to use slot selector, yet the slot selector button was not enabled. He finally clicked the “ObjSlot [1]” button, selected “ObjSlot” and then clicked on the slot selector button and got a core dump.

It turns out a change had been made to the code which first required the user to select the type of slot (here, ObjSlot[1]), select the ObjSlot expression and then use the slot selector to select the slot. This seems like an unnecessary extra step, since the program can infer the type of slot based on the slot selected and then create the appropriate expression.

12:40pm UserA mentioned that he really likes having ruleset in one file as opposed to multiple files as was the case with the old rule language.

12:40pm UserC did not seem to realize that the ruleset was one file, as became clear while saving a ruleset from the rule editor. He wanted to name the file to something specific to the rule rather than the ruleset as a whole.

This happened in the context of the first time the ruleset was saved. In this case, since the ruleset was newly created, the user is prompted to provide a name for the heretofore unnamed ruleset. The problem seemed to be because they invoked the save option from the rule editor and was despite the fact that I had changed the “File->Save” to “File->Save Ruleset”. I made this change to avoid this problem. I am not sure there is more I can do from an interface perspective. I think is a strong argument for good user documentation.

12:45pm UserB feels that entering date/times is difficult.

This should be changed to allow the user an easier way of entering date/times. The current syntax is @“<string>” where <string> is parsable by the date/time parser. Some thoughts:

- It would be good to drop the@“<string>” from the user’s perspective. Perhaps using a font/color/style change could be used to should this as a date/time.
- The syntax allowed by the date/time parser is, in general, intuitive. E.G., “May 5, 1997”.
- There are examples which are less intuitive (e.g., “Current Timestep”), but not too difficult

if remembered. In these cases, I would like to see some rule language-based (as opposed to date/time language) short hand versions allowed. For example, NOW for @"Current Timestep", PREV for @"Previous Timestep", etc.

- There are even less intuitive examples, like @"Current Month 5, Previous Year". While this provides a great deal of flexibility, it can be difficult to remember, create and read.
- A partial solution to the above issues would be to added explicit palette support. This could include a list of frequently used date/times as well as a series of dialog items which could be used to create date times.

12:45pm UserA mentioned that in-line editing mistakes cause the editor to close and revert to the pre-mistake state. It would be better to keep this editor open and allow user to correct mistake.

He's right. This seems to be a good general rule which is to always allow the user to correct their mistakes from the point at which the mistake was made. This done everywhere else (as least that I can think of) in the editors, except here.

12:45pm UserA wasn't sure what to use as an antecedent for an IF statement (on the LHS of an assignment statement). He thought he could enter a date directly and it would be automatically compared to the current date. E.G., IF (@"January") THEN... would mean if the current month is January, then the antecedent is true.

This seems like a nice thing, although it goes against the strong typing conventions of the language. The alternative is to enter a more explicit antecedent: IF (@"Current Month" == @"January") THEN... This might is a little more to type and perhaps a bit more difficult to read for those who know what the former example means. On the other hand, a new user might not know what the former example means.

There is a bug which currently allows a user to *paste* a date/time into an IF's antecedent. I don't think this would lead UserA to think this was valid since a ruleset with a date/time as an antecedent would not validate.

12:50pm UserB copied/pasted one date/time to another and added an ELSE to the IF. UserA helped him on this one.

12:55pm UserC wants to create an external function. He first looked on palette, then on rule editor, then at group editor on File menu. He mentioned that he needed a "sharing group". I told him about Utility Groups. He created one and added an external function.

This was encouraging since I had created the Utility Group for just this purpose. There is a bug which only allows one group editor per ruleset to be opened at a time.

1:05pm UserC opened an external function and wanted to copy contents of flat file into it.

OpenLook window manager doesn't seem to allow it, although the Motif window manager does. They were able to do it, but it wasn't as straightforward as it should be. Ideally the window manager and Galaxy would use a common buffer and allow this using the copy/paste keys. Since this doesn't seem to be an assumption that we can make, an import/export file option for this editor would be useful here.

1:15pm UserC copied a Tcl procedure into the body of the external function, but didn't realize that he had to strip off the procedure's header and trailing brace.

This is necessary since I create a Tcl function from the external function's body. This is not something that they would know and probably should be documented since it is unclear that

any other approach would work. One consequence of this is that if the user didn't remove the extraneous code, there would be no error until run-time since the Tcl code is not interpreted until run-time.

1:20pm UserC wanted to add a function call to the first consequent of an IF on assignment statement's RHS. He went to palette first and couldn't find it. He went back to the consequent and tried to type it directly via the in-line editor. I asked him where this function was and he realized that it needed to be created, but still wasn't sure if a function could be referenced before it had been created.

This is another example of the top-down restriction of the editing environment. Perhaps a future version could allow in-line editing and if a function call were entered and if that function didn't currently exist, the user could be prompted to create on. Perhaps just a return type would be required at this point and the user could provide the body at a later time. This latter restriction would be necessary to insure that the function returned the correct type.

1:30pm UserC created this function and another and said that he wanted to have these functions just return numbers so that he could see how things worked. I had to tell them how to return a value in Tcl.

Returning things from an external function is not straightforward, because the external function is written in Tcl and the returned value(s) must be parsable by the rule language. E.G.:
- ***return \"string\"*** returns a string.
- ***return "string"*** returns the identifier string which is likely to be an unbound variable. If it is bound, this could cause some unpredictable results.
- ***return "1.0 \[\"acre-feet/month\"\]"*** returns 1.0 [acre-feet/month].

This is probably necessary given that external functions are really tying together the two languages. For example, "\[" and "\]" are needed to keep Tcl from trying to interpret what is between the brackets.

1:30pm UserC, after he had created two very simple external functions (i.e., they just returned a value and would only be used to test that everything was working), said he wanted to have his rule call them. He first tried to do this using the in-line editor and typing just the name of the function (i.e., no parentheses). I asked him what he was doing and he said he was trying to call the function, like in Tcl. I told him that this was not correct and then he said that he needed to prepend a "$" to the function name in order to get the value that comes from the function invocation. He then asked "where does Tcl leave off and new language start?" I told him and he finally realized and used the palette to add the function call.

There was obviously some confusion about where Tcl and the rule language should be used. Although the errors would have been caught during validation, it is somewhat troubling that this confusion exists. Removing the external functions would remove this problem, but then it would add a worse problem of requiring the users to translate their entire Tcl-based rules into the new language at one time. This might be another case for a good tutorial in which this issue is specifically addressed.

1:40pm UserC loaded a ruleset from *RiverWare's* "Policy->Load Ruleset..." menu option, rather than using one of the ruleset's editors.

This could cause some confusion in that the user thought he was loading the ruleset he was editing, which he is. The only problem is that he is really loading the last saved version of the ruleset he is editing. Yet, I don't want to prohibit the loading of a ruleset which is not being edited, since I really don't want to require the user to open a ruleset to load it. It could be argued

that since the currently open ruleset does not have the “Ruleset Loaded” message/color displayed, the user should realize that he hasn’t loaded this ruleset, although I can see how this could be easily missed. Again this might be a case for good documentation that addresses this problem.

This might be a case for problem-based documentation in which user tests are done to determine the problems that are most likely to be encountered during use of the system. Of those which cannot or should not be changed (since the problems they would create would be as significant as those they solved), a set of problem-specific examples/mini-tutorials could be developed which specifically address these problems and show by example/discussion that these problems exist, what happens if they are encountered and how to avoid them.

1:40pm UserC tried to run the simulation, but he got a rule execution error, since he had not set the external function’s return type. He corrected this and the simulation ran properly.

External functions have return types which cannot be inferred as is the case with rule language functions. As a result the user is required to set this explicitly. This can be easy to forget. Probably the default should be set to NUMERIC, since this is the typical return type for an external function. One thing to note is that if the return type is incompatible with what is actually returned, an error will be generated.

1:40pm UserC said “I like somethings about it”. He also said that he would have liked to have been given a syntax overview before formulating the rules (as they did yesterday in the conference room).

1:45pm UserC wanted to add arguments to external functions. Here is the sequence:

1. Typed just argument names. Got parse errors.
2. Added argument types, but typed these in lowercase. Also, left off the commas. Got parse errors. He asked if spaces were okay and I told him that they were needed between the argument type and name. He then asked if commas were necessary, and I told him to try it. He put in the commas and got another parse error.
3. Changed argument types to uppercase. This worked, although when he hit the carriage return, he got no feedback that this worked.

Function argument editing needs some serious work. In an effort to get the editors working, I didn’t put the work necessary into this portion of the GUI. As a result, there is no structure editing support here and the only feedback that is given is negative feedback. At a minimum, the palette should be augmented to help here or (if this violates the non-editor specific nature of the palette) the function editors themselves should be changed to provide explicit support. I suppose the moral of the story here is to not mix structure and non-structure editing too much. The in-line editor might be a reasonable exception, since it is self-contained and provides appropriate feedback.

The fact that I didn’t do everything I wanted might actually be a good thing in that this is not an unusually occurrence in a time/budget constrained project. As such, it helped to bring out some problems that can occur in these situations and help point out some categories of problems that should be avoided.

1:50pm UserC wanted to PRINT a function’s arguments, but didn’t know how. I told him to look at hidden function items. He did and found out how to PRINT the arguments.

The use of hidden function items has some drawbacks, although I think that overall the benefits outweigh the drawbacks. Documentation again.

1:55pm UserC, after he had changed the functions to have them take arguments, needed to put the new functions in place of the old ones. This didn't work since there appears to be a bug regarding the return types of newly created functions. Otherwise this works fine as long as the return types of the functions are not changed. If they are, the IF's consequences could become incompatible and the user would be required to clear one of them in order to re-enable the palette.

This needs to be addressed. Either the palette must be enabled at all times so that incompatibilities can be fixed easily or these incompatibilities must be prohibited. On the other hand, this kind of thing is a result of the user making changes which are reasonable, as are the resulting incompatibilities. Perhaps when changing a function's return type, a warning should be displayed that this change will cause an incompatibility and allow the user to either clear the calls to these functions or cancel the return type change.

2:00pm BUG - UserC is updating expression, but the palette is not being updated.

2:00pm BUG - UserC is losing expressions.

2:00pm UserC turned on function diagnostics and viewed output.

2:00pm UserC spent the rest of the time working on his Tcl code in his external functions.

3:00pm BUG - UserC's dates are passed to Tcl functions as "Current Timestep" rather than the date to which they resolve.