A C++ Class for Rule-Base Objects

WILLIAM J. GRENNEY

Professor, Civil and Environmental Engineering, Utah State University, Logan, UT 84322-8200

ABSTRACT

A C++ class, called Tripod, was created as a tool to assist with the development of rule-base decision support systems. The Tripod class contains data structures for the rule-base and member functions for operating on the data. The rule-base is defined by three ASCII files. These files are translated by a preprocessor into a single file that is located when a rule-base object is instantiated. The Tripod class was tested as part of a prototype decision support system (DSS) for winter highway maintenance in the Intermountain West. The DSS is composed of two principal modules: the main program, called the wrapper, and a Tripod rule-base object. The wrapper is a procedural module that interfaces with remote sensors and an external meterological database. The rule-base contains the logic for advising an inexperienced user and for assisting with the decision making process. © 1993 by John Wiley & Sons, Inc.

1 INTRODUCTION

States in the Intermountain West support expensive programs for highway snow and ice removal. For example, the state of Utah expects to spend over $7,000,000 per year on state highways for winter maintenance. The timing of the dispatch and release of road crews is important to safety and costs. If the crews are dispatched too late or released too early, driving hazards could develop. If the crews are dispatched too early or released too late, excessive costs could be incurred. Decisions regarding the deployment of personnel and heavy equipment are made by maintenance foremen and supervisors. Data available to them include local and national weather forecasts and real time data from remote sensors, as well as personal observations.

Received May 1992
Accepted November 1992

© 1993 by John Wiley & Sons, Inc.

Scientific Programming, Vol. 1, pp. 163–175 (1992)
CCC 1058-9244/93/020163-13

The purpose of this study was to develop a rule-base decision support system (DSS) to assist maintenance foremen perform their duties during winter storms. A secondary objective was to provide a training device for inexperienced personnel. The study involved two concurrent endeavors:

1. Selection or development of a tool for implementing a rule-base DSS
2. Development of a prototype application for winter highway maintenance utilizing the tool

Most of the study effort went into the development of the rule-base DSS tool that is described in this paper.

2 IMPLEMENTATION STRATEGY

The first activity undertaken to design the DSS was the establishment of a panel of experts. Four foremen from highway maintenance departments in Colorado and Utah were invited to form the panel. They ranged in education from high school

to 4 years of college engineering. All four had lengthy experience making decisions regarding snow plow and deicing operations. During the first meeting the experts listed the decisions that they make during a variety of winter storm events, and identified the data that would be nice to have available in order to decide on a particular course of action. They considered the following features essential for a useful DSS:

1. Lead a novice user through a step-by-step process to reach a reasonable conclusion
2. Display tables and graphics quickly and concisely for experienced users
3. Provide optional supplemental information to assist with the interpretation of displayed data
4. Interface with third party software to access remote sensors and data bases
5. Operate on a computer platform readily available to maintenance foremen

This last feature dictated the use of IBM 386 class microcomputers because they are affordable or already available at highway maintenance sites. Also, the software interface provided by the manufacturer of the remote sensors operates under DOS. However, we also considered portability to be an important feature. MS windows, OS2, and UNIX are encroaching on DOS, and we would like to adapt to these operating systems as they become more prevalent.

The second activity undertaken to design the DSS involved the selection of a modeling technique (language or shell) for the tool. The rule-base structure was selected as the basis for the DSS for several reasons. It is a natural way to represent the decision making process, it is a better way than most to capture the thought processes of experts, and it can be expanded and updated with a minimum of reprogramming effort [1–3]. We decided to evaluate three techniques: (1) a commercial shell, (2) Prolog, and (3) C++. Two graduate students went to work on two separate prototype projects. One used the INSIGHT 2+ commercial shell [4] and the other used Turbo Prolog [5]. I developed a prototype using Turbo C++ [6]. We compared progress weekly during the evaluation period.

Work with the shell was pursued for about 2 months and then abandoned. The shell worked very well for quickly establishing rules and it provided an impressive user interface. However, we had difficulty interfacing the software for the remote sensors. This software was an undocumented stand-alone package composed of some complex DOS batch files that executed Pascal programs. The software changed environmental variables, dominated memory, and frequently switched the screen between text and graphics modes. The software was proprietary and we did not have access to the source code. In the final analysis, we just did not have the expertise necessary to interface the remote sensors with the shell. The shell did give us, however, a good initial set of rules for the other two projects.

The second student had no previous experience with Prolog. After climbing the learning curve, he came to prefer Prolog over C and Fortran. He utilized Borland's Turbo Prolog Toolbox [5] to develop effective pull-down menus. He coded a module that responds to a menu item and automatically accesses a national weather data base [7]. He developed routines to select data from the data base and display it in graphical formats. He made an initial attempt to interface the sensor manufacturer's software package and was not successful. After the difficulties encountered during our first experience, we elected not to spend additional time trying to interface the sensors, but instead to complete his project as a prototype interface to the national weather data base [8].

During my evaluation project, I found it difficult to adjust from traditional structured programming to the object oriented programming (OOP) approach advocated by C++ developers. However, it was worth the effort. OOP is intuitively consistent with the event-driven paradigm that is becoming increasingly popular, and which I found to be an effective way to design the DSS. Although we still had trouble interfacing with the remote sensors, many alternatives for work-arounds were available. We decided to select C++ over a commercial shell and Prolog. The main reasons follow:

1. The commercial shell was effective for quickly constructing a prototype. However, learning to use it for a practical application involving remote sensors appeared to be almost as difficult as learning a lower level language, and the shell lacked procedural programming capabilities. A commercial shell would not be as portable to other platforms as C/C++. Also, commercial shells often have a royalty fee associated with each run-time license.

2. Turbo Prolog was found to be a very powerful language for this application, especially when used in conjunction with Borland's Toolbox. Although popular in other parts of the world, Prolog appears to be declining and losing support in the United States. For example, Borland discontinued support for Turbo Prolog. We did not think that Prolog would be as portable to other platforms as C/C++.
3. Momentum is growing in support and extensions for C++. Borland's C++ provides a framework for developing object oriented programs. OOP is intuitively consistent with the event-driven paradigm that is becoming increasingly popular.

It should be emphasized that this evaluation of three products was far from scientific or comprehensive. It focused on one application, and in particular on portability and the interfacing of procedural modules. At the very least it was biased by my background as a C programmer. However, in summary it is safe to say that neither the commercial shell nor Prolog offers sufficient advantages to switch from the C/C++ family for this application.

Another consideration played an important part in the decision. The experts had trouble formulating their ideas directly into typical antecedent-consequence type rules in conventional formats. However, they were quite effective in developing rules by means of a three-step procedure. The first step was to make a list of the important actions (consequences) that they would perform during a winter storm event. The second step was to make a list of variables (types of data) that were useful in deciding which action to take. The third step involved the development of a decision matrix for associating actions with variables and their values. The decision matrix defines the rules for the application. An example of the rule format is presented later in this paper.

The three data sets assembled by the above procedure completely define the rule-base. We wanted to develop a preprocessor that would automatically convert the three data sets into a single rule-base that could be efficiently input to a DSS of our design. It seemed like a good idea to translate directly from formats that the experts were able to construct and comprehend, to a rule-base file that the DSS could efficiently read. By doing so, the experts could quickly see the results of a change to one of their files without an intermediate interpretation by a knowledge engineer. All of the knowledge engineering goes into the formation of the three data sets which, for this case, were easily understood by the experts. We found that C/C++ was an effective language for developing both the preprocessor and the DSS.

3 PROGRAM DESIGN

Considering the list of essential features for the DSS previously listed, some are more suitable to a rule-base design and some to a procedural design. The decision making and informative aspects of the application lend themselves to the rule-base approach. The sensor interfacing and data manipulation aspects are best suited to the procedural approach. We wanted a hybrid tool that provides the flexibility of both approaches. We distinguished between two fundamental approaches: (1) formulating the application in a declarative architecture that utilizes and coordinates procedural modules and (2) formulating the application in a procedural architecture that utilizes and coordinates rule-base objects. The commercial shell and to some extent Prolog are declarative approaches that permit interfacing with procedural modules. We found significant problems with data transfer among modules. Because we were much more confident in our procedural than declarative programming skills, we decided to design a tool composed of a procedural "wrapper" that has the capability to instantiate multiple rule-base objects. Because our rule-base object obtains knowledge from three data sets, we named it "Tripod." Information exchange (data and commands) is accomplished by messages sent between the wrapper and the Tripod object.

3.1 Rule-Base

Figure 1 illustrates the DSS application process. The terminology used in the following discussion evolved during interviews with the panel of experts. Terms were selected that provided the best communication with the experts. Three ASCII files, shown at the left of Figure 1 are prepared by the experts and the developer. These files are described in greater detail later in this paper.

3.1.1 The Variables File

A variable is a quantifiable characteristic of the system. The state of the system at any time is de-

fined by the values of the variables. The variable file (identified by the extension .IDV) contains a list of unique identification symbols and associated information for each variable, such as the type of the variable (i.e., integer, float, logical, menu, or string). Each variable also has a question associated with it that, when needed, will be displayed on the screen requesting a value from the user.

3.1.2 The Actions File

An action is a consequence that is invoked when an associated rule fires (e.g., evaluates true). The actions file (identified by extension .IDA) contains a list of unique identification symbols and associated information for each action, such as the type of action to be performed. An action may display text, display graphics, or perform an operation. A variety of operations may be performed by an action, for example, manipulating the values of variables, sending a message to the wrapper, and branching to a specified rule in the rule-base.

3.1.3 Rules and Tests

A rule is an IF-THEN statement specifying the appropriate action for a particular state of the system, as defined by the values of the variables. It has the standard form:

```
IF (variable_1 == ruleValue_1) AND (variable_2 <= ruleValue_2) AND .... THEN (action_1)
```

The antecedent is made up of "tests," shown in parentheses. For a test, if the value of the variable (i.e., variable_1) satisfies the comparison (i.e., equals ruleValue_1), then the test is true. If all of the tests for a rule are true, then the rule is true and the action is invoked. If any one of the tests is false, then the rule is false. The rule file (identified by the extension .IDR) contains a list of unique identification symbols and associated information for each rule. Among other things, each rule has a list of tests that can include variables, actions, and other rules.

3.1.4 The Preprocessor

A preprocessor performs error checking on the three files and produces a rule-base file that can be efficiently read by the Tripod object. Changes to the rule-base file may be made only through the variables, actions, and rules files.

3.1.5 The Fact Data Base File

Figure 1 also shows a "Fact Data Base" file. This file contains initial values for variables in the rule-base. It is an ASCII file with a list of variable identification symbols, each followed by its initial value. The Fact Data Base may be updated any time prior to the instantiation of the Tripod object. Variables not listed in the Fact Data Base are automatically initialized to an "unknown" state by the object.

3.2 The Wrapper

Figure 1 shows the wrapper interacting with a Tripod object and an auxiliary process (such as a remote sensor). The wrapper is a procedural main program that instantiates and coordinates Tripod objects as well as performs auxiliary processes. The wrapper is the master of the application: it

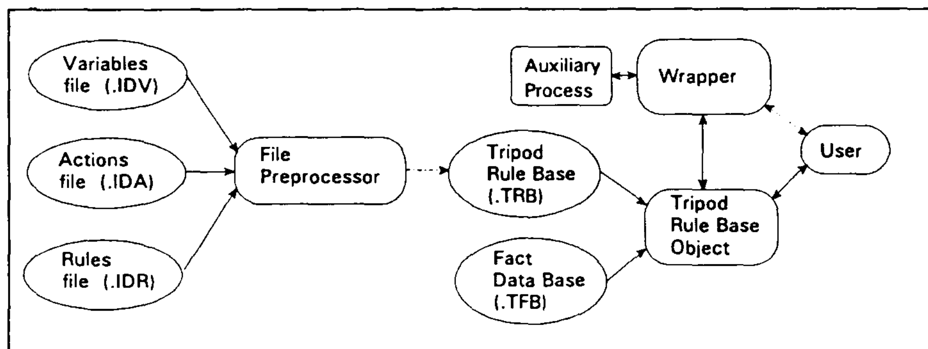


FIGURE 1 : The Decision Support System application process.

communicates with Tripod objects by means of messages and gathers information to respond to the messages by means of procedural functions (auxiliary processes). The wrapper is tailored for the rule-base, or set of rule-bases, in a specific application. The wrapper must know how to respond to messages it receives. Messages are integer values. For example, action_message "2" from the Tripod object requests data from a remote sensor buffer. The wrapper often responds by sending a message containing the data back to the Tripod object.

3.3 The Tripod Object

When a Tripod object is instantiated by the wrapper, it fills in data structures from the Rule-Base file. It then initializes appropriate data member values from the Fact Data Base file, and returns control to the wrapper. The object is dormant until it receives a message from the wrapper to begin testing its rule-base at a specified rule. Rule testing is conducted by a member function in the Tripod object that is described later in this paper. Rules are tested in sequence until one fires. When a rule fires the associated action is performed by another Tripod member function, a "rule_fired" message is sent, and program control is returned to the wrapper.

3.4 Program Procedure

3.4.1 Overview

Figure 2 illustrates the program procedure as a series of communication events between the wrapper and the Tripod object. The circled numbers in Figure 2 correspond to numbers in parentheses in the following discussion.

Starting at the top left of Figure 2, the user executes the wrapper and the wrapper instantiates a Tripod rule-base object (1). The object's constructor loads the Rule-Base file, initializes variables from the Fact Data Base file, and returns control to the wrapper (2). The wrapper sends a message to the Tripod object to begin testing rules in sequence, starting with a designated rule (3). The rule testing algorithm in the object evaluates the tests associated with the current rule. Most tests will request a response from the user as indicated in Figure 2.

Any time the Tripod object requests a response, the user has the option to send a message to the wrapper via the object. When this happens the

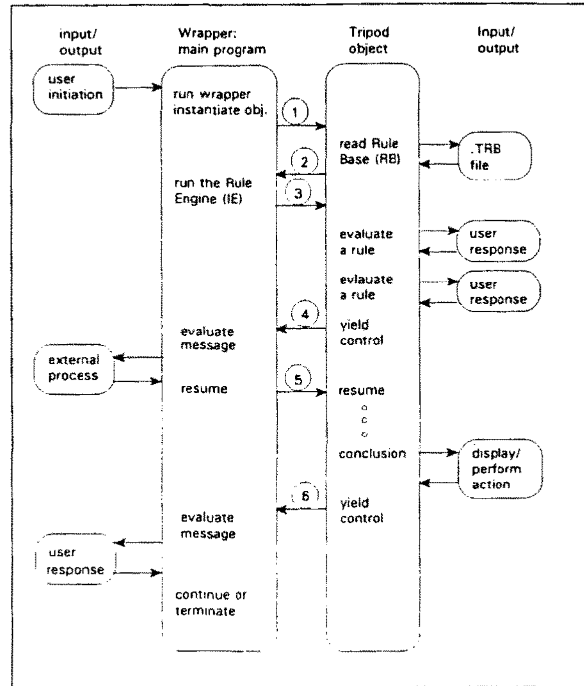


FIGURE 2 Program procedure.

object stores its current status, sends the message, and returns control to the wrapper (4). After the wrapper responds to the message, which might be the execution of an auxiliary process, for example, it sends a message to the object to resume from where it left off (5).

When a rule is true (i.e., all of its tests evaluate true) the associated action is invoked, and a "rule_fired" message is sent and control is returned to the wrapper (6). The wrapper typically requests the user to decide whether to continue or to terminate the run.

3.4.2 The Rule Engine

The Rule Engine is the member function of the Tripod object that evaluates the rules. It takes as a parameter the order number of a rule in the rule-base. It starts by evaluating the first test in the list of tests associated with that rule. If the first test evaluates true, it goes to the second, and so on. If all of the tests are true, the Rule Engine calls the function `execAction()`. This function takes as a parameter the pointer to an action, and performs the operation specified by the action. For example, it might replace the value of one variable with the value from another, or it might just display text on the screen. After `execAction()` returns, the Rule Engine stores the current status of the Tripod

object, sends, a "rule_fired" message, and returns control to the wrapper.

If a test evaluates false, the Rule Engine discontinues evaluating tests for the rule, marks the rule as false, and goes on to the next rule in the rule-base. This process continues until a rule is encountered in which all tests evaluate true.

Three types of tests are permitted: variable, rule, and action. The variable type test provides the symbol for the variable and a value (or range of values). The Rule Engine first checks to see if the variable has been previously evaluated and if not calls a member function `query()`. This function displays the question associated with the variable, and compares the user's response with the test value(s). It returns a true or false flag to the Rule Engine depending on the results of the comparison.

A rule type test provides a pointer to another rule. The Rule Engine calls itself to evaluate this rule. If the rule is true then the test is true, otherwise the test is false. The Rule Engine operates recursively on nested rules to any reasonable depth.

An action type test provides a pointer to an action. The Rule Engine checks to see if the action has previously been invoked (i.e., associated with a rule that has already fired), and if not begins testing all of the rules associated with the action. If one of these rules fires, then the action is invoked and the test is true. If none of these rules fire, then the test is false.

Branching is handled by a special operation in an action. When an action is invoked by `execAction()` it may perform several types of operations. One type is to branch to a specified rule. This is accomplished by returning a message to the Rule Engine directing it to leave off what it has been doing and begin evaluating the specified rule. For example, if a nested rule fires and its action specifies a branch to another section of the rule-base, the Rule Engine does not finish evaluating the higher rules in the nest.

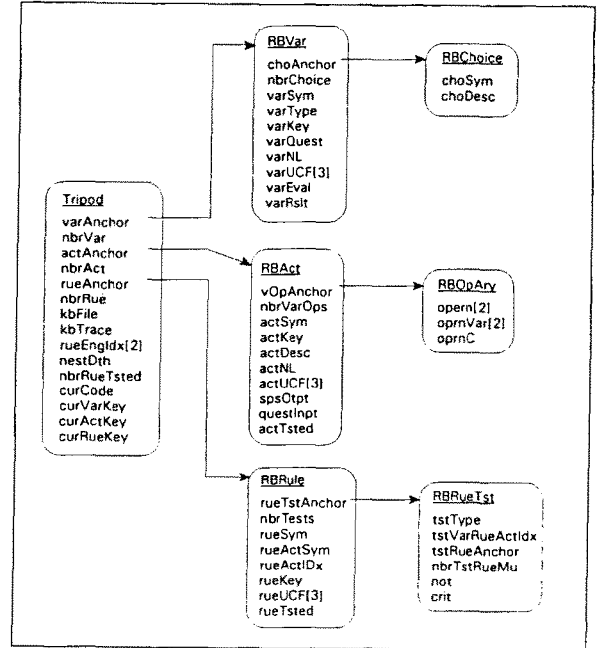


FIGURE 3 Tripod data structures.

4 TRIPOD DATA STRUCTURES

4.1 The Tripod Class

Continuity is maintained by continually updating the Tripod data structures. The Tripod object retains the results of all previous operations. For example, it records which rules have fired, which actions have been invoked, which variables have been evaluated, and what values have been assigned. This is facilitated by encapsulating the data for the actions, variables, and rules separately.

Figure 3 shows the most important data members in the class structure. The Tripod class contains `varAnchor`, `actAnchor`, and `rueAnchor`, which are pointers to lists of variables, actions, and rules, respectively. The data members `nbrVar`, `nbrAct`, and `nbrRue` are the number of entries in the respective lists. The data members `kbFile` and `kbTrace` are file handles for the rule-base input file and an output file to record a trace of the steps performed by the user. The other data members are for control purposes:

1. `rueEngIdx` [2] records the indices of the principal rule (the highest in the hierarchy of nested rules) and the current nested rule being evaluated.
2. `nestDth` is the depth of the current rule being evaluated within a set of nested rules.
3. `nbrRueTsted` is the number of rules that has been tested during the current run.
4. `curCode` is the latest message sent to the wrapper from the Rule Engine.
5. `curRueKey` is the key (integer code) associated with the current rule being evaluated in by the Rule Engine. The key is assigned by

the developer in the rules file. It is often used to indicate a help or supplemental information file (i.e., 54.txt).

6. curVarKey is the key (integer code) associated with the current variable being evaluated by the Rule Engine. The key is assigned by the developer in the variables file.
7. curActKey is the key (integer code) associated with the current rule being evaluated by the Rule Engine. It is assigned by the developer in the actions file.

4.2 The RBVar and RBChoise Classes

The RBVar (Rule-Base Variables) class contains data for each decision variable. The data member choAnchor points to a list of menu choices for a menu type of variable, and nbrChoice is the number of choices in the list. The RBChoice class contains the information for each menu item: choSym is a unique character string identification symbol and choDesc is a character string description for the item. varSym is a character string for a unique identification symbol for this variable. varType is the type of the variable that may be one of the following: menu, logical, string, integer, or floating point. varKey is the integer code assigned by the developer in the variables file to specify an auxiliary process.

An auxiliary process is normally a function programmed by the developer to accomplish a specific task: for example, reading a remote sensor, instantiating another Tripod object with a different rule-base, or displaying a text file on the screen. A developer may wish to use key "54" to display text file 54.txt on the screen when the user requests help with the query for a particular variable. The developer assigns the value 54 to the key for the variable in the variable file, and then programs a function for the wrapper to respond to a "variable key" message having the value 54 by displaying 54.txt.

varQuest is the question for the variable and varNL is the number of lines in the question. varUCF [3] stores three arbitrary coefficients for the variable. Originally these coefficients were going to be used to represent uncertainty. However, this feature has not yet been implemented and the coefficients are unused. varEval is a flag indicating whether or not this variable has been previously evaluated. If it has been previously evaluated, varRslt stores the value. varRslt is a data union that stores a logical (Boolean), string, float, menu, or integer depending on the type of variable.

4.3 The RBAct and RBOpAry Classes

The RBAct (Rule-Base Actions) class contains data for each action. The data member vOpAnchor points to a list of operations that the action is to perform on decision variables, and nbrVarOps is the number of operations in the list. The RBOpAry class contains information for the operation. opern [2] stores two character symbols indicating the type of operation to be performed. oprnVar [2] contains the two operands. Three types of operations are currently available, and they are described in the next section.

Other data members in RBAct include actSym, actKey, and actUFC [3], which are analogous to varSym, varKey, and varUCF [3] previously defined. actDesc is a character string with the description of the action, and actNL is the number of lines in the description. spsOtpt is a flag to suppress display of the description. actTsted is a flag to indicate whether or not the action has been previously invoked. Setting this flag suppresses subsequent executions of this action. questInpt is a flag modifying the action. Three flags are permitted at this time, and they are described in the next section.

4.4 The RBRue and RBRueTst Classes

The RBRue (Rule-Base Rules) class contains data for each rule. The data member rueTstAnchor points to the list of tests for the rule, and nbrTests is the number of tests in the list. rueSym is a character string containing the unique identifier for the rule. rueActSym is the symbol of the action associated with this rule, and rueActPtr is a pointer to it. rueKey, rueUCF [3], and rueTsted are analogous to actKey, actUCF [3], and actTsted.

The RBRueTst class contains information for each test. tstType is a character indicating the type of test: "V" for variable, "R" for rule, and "A" for action. tstVarRueActIdx is the identifier of the variable, rule, or action for this test. tstRueAnchor is a pointer to a list of pointers to rules nested in this test. nbrTstRueMu is the number of nested rules in this test. not is a flag indicating whether NOT should be applied to the results of this test (i.e., true becomes false). crit is a data union for a Boolean, string, float, or integer. It contains the values that will cause the test to evaluate true.

5 PROTOTYPE APPLICATION

Decisions regarding the deployment of personnel and heavy equipment for snow and ice prevention and removal are made by the maintenance foremen in geographical districts. Districts range in size from a few hundred to several thousand square kilometers. Data available to the foremen include local and national weather forecasts, real time data from remote sensors, and personal observations. During a winter event, foremen rely upon these data in varying degrees and upon personal rules of thumb for dispatching crews and equipment, and for releasing crews near the end of the event.

During the first three meetings with the panel of experts, the scope of the problem was discussed and terminology was established that was meaningful to the members of the panel. The objective of the study was focused on preparing a simple prototype of a DSS that would help foremen make decisions about the dispatch and release of road crews during a winter event. The prototype should provide table and graphic displays of data from remote sensors, and provide a step-by-step advisor for novice users [9].

The importance of terminology used for communicating with the panel of experts should not be underestimated. Terms were established by the panel defining three categories of information: variables, actions, and rules. Forms were developed to assemble information in these three categories. These forms were the basis for the input files for the preprocessor.

5.1 The Variables File

The variables file contains the input data for the variables. Table 1 shows the input format for each variable in the file. A unique symbol for the variable is enclosed in parentheses. The symbol may be up to 20 characters long. This is followed by the declaration of the type of variable: menu (M), string (S), logical (L), integer (I), or floating point (F). A menu type declaration must be followed by

Table 1. Input Format for Variables File

```
(variableSymbol)   M3  2  1  0.1  0.2  0.3
This line contains the question for the
user.
choiceSymbol_1     First menu choice
choiceSymbol_2     Second menu choice
choiceSymbol_3     Third menu choice
```

the number of items in the menu list, as shown above. The next field contains the integer "key" code (2). This code is passed to the wrapper with a message from the Rule Engine when an external process is requested for the variable. The next item in the first row is an integer specifying the number of lines in the question. The last three fields (0.1, 0.2, 0.3) were originally intended for coefficients quantifying uncertainty, but this feature is not yet implemented.

The second row contains the question that will be displayed for the user when this variable is encountered in a rule test. If the variable is a menu type, as shown in Table 1, the remaining rows contain the choices for the menu. If the variable is an integer of float type, only one row follows the question, and it contains the minimum and maximum permissible values for the variable.

Table 2 contains a partial listing of the variables identified by the experts. A key of "1" signals the wrapper to obtain and display information from the ScanCast weather forecasting system [10]. This procedure requires accessing a remote data base over telephone lines. A key of "2" signals the wrapper to obtain current meterological data from remote sensors at a highway station. This information includes air temperature, pavement temperature, wind speed, precipitation, relative humidity, and other pertinent data.

5.2 The Actions File

The actions file contains specific actions to be performed as a consequence of the state of the system. An action may be a screen display giving recommendations to the user, an operation to be performed on a set of variables, or a message to the wrapper to execute an external process. Table 3 shows the input format for each action in the file. A unique symbol for the action is enclosed in parentheses, followed by the key code. Three flags are permitted following the key:

1. /S suppresses the screen display of the action description
2. /Q signifies "questionnaire" and will force all tests of a rule having this action to be evaluated. Normally, the evaluation of tests within a rule will be discontinued at the first false.
3. /O indicates that this action is to perform operations as specified by codes following the description line(s).

Table 2. A Partial Listing of the Variables File (.IDV)

```
(expectPrecip)                      M3   2   1
Is there a possibility of precipitation in the next 3 hours?
rain            There is a possibility of rain in the next 3
                hours.
snow            There is a possibility of snow in the next 3
                hours.
no              No possibility of precipitation in the next
                3 hours.

(expectDrift)                   L    2   1
Is there a possibility of drifting in the next 3 hours?

(snoRateNow)                    M3   1   1
What is the rate of snow fall now?
light           Light (less than 1 in/hr).
medium          Medium (between 1 and 2 in/hr).
heavy           Heavy (greater than 2 in/hr).

(hwyPriority)                   I    0   1
What highway priority are you considering (1 to 4)?
1 4

(crewOnDuty)                    M4   0   1
Is there a crew on duty?
noCrew          No crew is on duty.
nightCrew       A normal night crew is on duty.
skelCrew        A skeleton crew is on duty.
maintCrew       A maintenance crew is on duty.

(stormDirection)                M4   2   1
The storm approaches from which direction?
southEast       Storm from the southeast (moisture).
northWest       Storm from the northwest (winds).
west            Storm from the west (moisture).
southWest       Storm from the southwest (severe).

(timeOfDay)                     F    0   1
What is the time to the nearest half hour (24 hour clock)?
0 24.5

(specialEvent)                  M4   0   1
What type of special event is scheduled?
sport           Major sports event.
conv            Major convention.
holiday         Holiday celebration.
polit           Major political event.
```

The next field in the first row specifies the number of lines in the description, and the last three fields are place holders for future development. The second row contains the description of the action. This text will be displayed to the user when this action is invoked unless the /S flag is set. If an operation flag (/O) has been set, then additional rows are needed to specify the operations to be performed. Three types of operations are currently available:

Table 3. Input Format for Actions File

```
(actionSymbol) 57/S/Q/O  1  0.1  0.2  0.3
This line contains the description of the
action.
=V  variableSymbol_1     variableSymbol_2
=C  variableSymbol_3     constantValue
B   ruleSymbol
```

1. "=V" means set the value of the first variable equal to the value of the second.
2. "=C" means set the value of the specified variable equal to the value of the constant.
3. "B" means branch to the specified rule.

Table 4. A Partial Listing of the Actions File (.IDA)

```
(Wait)                  0  1
No decision necessary at this time.

(AlertCrew)             0  1
Alert crews for possible dispatch in 2 to 3 hr.

(EmergencyAlert)    0  2
Emergency condition expected, alert contractors, and/or highway
patrol.

(DispatchSklCrew)   0  1
Dispatch a skeleton crew.

(DispatchCrewSnd)   0  1
Dispatch a standard crew with sand trucks.

(DispatchCrewP/S)   0  1
Dispatch a standard crew with plow and sand.

(EmergencyAct)          0  1
Implement emergency actions: contractors, state patrol, etc.

(ReduceToSkel)          0  1
Reduce to skeleton crew.

(ReleaseCrew)           0  1
Release Crew.

(AlertRelCrew)          0  1
Alert relief crew for possible dispatch in 2 to 3 hr.

(DispatchRelCrew)   0  1
Dispatch a relief crew to replace a standard crew.
```

Table 4 contains a partial listing of the actions identified by the experts. Each action is made up of a unique symbol and an expanded description. No keys were needed for this application. Blank fields in the input file are interpreted as null by the file preprocessor.

5.3 The Rules File

Rules associate the state of the system, as characterized by the values of the decision variables, with the actions to be performed. Each rule contains a list of tests. A test may be based on a variable, an action, or another rule. In its simplest form, the rule-base may be represent by a decision matrix where each row is a rule and each column is a test. The elements in a row contain the values that will cause the tests to evaluate true. An action is associated with each rule, and when a specific rule fires, it is this action that is invoked.

The experts used a decision matrix to establish the rules for this application. Each row (rule) in the decision matrix was represented in the input file as shown in Table 5. A unique symbol for the rule is enclosed in parentheses, followed by the symbol for the action associated with this rule, the key code, and the three unused coefficients. The second row contains the description of the rule.

Each row following the description describes a test for the rule. The first two characters in each row indicate the kind of test. A "!" as the first character indicates that the result of the test is to be reversed (i.e., true becomes false) after evaluation. Three kinds of tests are permitted:

1. "V" means to test a variable against the specified criteria.

Table 5. Input Format for Rules File

```
(ruleSymbol)           actionSymbol  111  0.1  0.2  0.3
This line contains the description of the rule.
V   variableSymbol_1          criterionForTRUE
R   ruleSymbol_1              ruleSymbol_2  ruleSymbol_3 ...
!A  actionSymbol              ruleSymbol_4  ruleSymbol_5 ...
```

Table 6. A Partial Listing of the Rule File (.IDR)

```
(102)                          AlertCrew        0
Put a crew on stand-by for snow.
v     expectPrecip                   snow
v     snowStart                      2 6
v     crewNotified                   false
v     crewOnDuty                     noCrew

(103)                          AlertCrew        0
Put a crew on stand-by for rain.
v     expectPrecip                   rain
v     rainStart                      2 6
v     rainRateThen                   mist      drizzle rain
v     crewNotified                   false
v     crewOnDuty                     noCrew
v     pvmtFrzStart                   2 6

(109)                          DispatchCrewP/S      0
Dispatch crew with plow and sand.
v     expectPrecip                   snow
v     snowStart                      0 2
v     snoRateThen                    light     medium heavy

(110)                          DispatchCrewSnd      0
Dispatch a standard crew with sand.
v     expectPrecip                   rain
v     pvmtFrzStart                   0 2
v     rainStart                      0 1

(111)                          DispatchCrewP/S      0
Dispatch crew with plow and sand.
v     expectPrecip                   snow
v     snoRateNow                     light     medium heavy

(112)                          DispatchCrewP/S      0
Dispatch crew with plow and sand.
v     expectDrift                    true
v     windStart                      lt_1           1_to_3
v     windSusThen                    10_to_20
v     SnoForDrift                    10 100
```

2. "R" means to test the following list of rules in the specified order.
3. "A" means to test an action. A list of rule symbols (associated with the action) may follow to dictate the way in which the action is to be evaluated.

Table 6 contains a partial listing of the input file for the rules developed by the experts. The experts would first select an action, then decide which variables were important for decisions regarding the action, and finally what values of the variables would cause the action to be taken. In Table 6, integer and floating point variables are followed by a range of values. The variable will test true for any value within this range, and false otherwise. Menu variables are followed by a list of choices. Selection of any one of these items will cause the variables to test true.

6 PRELIMINARY TESTING

Most of the testing effort went into exercising the procedures in the preprocessor, the wrapper, and the Tripod object. During this process many error traps and warnings were added in order to ensure consistency among the input files. The preprocessor prepares a list of all variable and action symbols not used in the rule file so that the developer can see if important elements have been inadvertently omitted from the rule-base. In the case where an action operates on two variables, the preprocessor checks to ensure that the variables

are of the same type. If they are menu type variables, the first variable must have at least as many menu items as the second and a warning is displayed if corresponding choice symbols are not identical.

The preprocessor also checks to see if the developer has used a rule in a test for that rule. For example, in a test containing several levels of nested rules, a developer could easily include the rule within a test for itself. Obviously such a mistake could result in run-time endless recursion, a situation that occurred on several occasions before this trap was added.

Next to displaying text for the user, the branch is the most useful action operation. We used it as a means to partition a rule-base. For example by evaluating a few rules first, we could branch to major sections of a rule-base to acquire detail information about a particular subject. Also, we could design the rule-base to skip over major sections of rules that are no longer relevant after a particular action has been invoked. Branching can be used to construct loops in the rule-base: however, we did not have an immediate practical need for this construct. Like nesting rules, using branches must be done carefully in order to avoid run-time thrashing. We have not devised a technique for the preprocessor to detect inappropriate branching.

During prototype testing, the wrapper and rule-base performed quickly and efficiently. The Rule Engine utilizes pointers extensively, and can evaluate more than 50 tests without an observable delay between display screens when operating on a 25 MHz 386 class microcomputer.

Preliminary testing of the prototype decision support system by the experts led to some adjustments to the existing rules and formulation of additional rules. The experts found the three data sets (variables, actions, and rules) to be a practical approach for constructing the rule-base.

The prototype was used in a case study for the Denver, Colorado area. The Colorado Department of Transportation maintains records of major storm events including weather forecasts prior to and during the storm, remote sensor data, and personnel and equipment allocations. A nonexpert was given the historical data from two storms, and asked to make decisions regarding the dispatch and release of maintenance crews. Using the decision support model, the nonexpert reached the same conclusions as the expert. The prototype was used for a small sample of relatively simple situations. Expansion of the rule-base and additional testing of much larger sample sizes would be needed in order to establish a useful decision support system.

Additional testing of the Tripod object was conducted for two more applications. One was a screening tool for transportation agencies to quickly evaluate the appropriateness of a variety of hazardous waste remediation technologies for a specific waste at a specific site [11]. This application demonstrated the coordination of three rule-bases, a data base, and a simulation model by means of a wrapper and three Tripod objects. We found it to be more convenient to assemble three rule-bases that could be instantiated as needed, than to assemble one large rule-base. The second was a conceptual application to reconcile nongraphical data with digital maps for use in Geographic Information Systems [12]. In this case, there was very little need to solicit information from the user during a session. The actions operate on the values of the variables according to a set of predefined conditions that are nearly the same for all nongraphical data files and digital map files. We found this approach to be a viable alternative to programming a complex system of IF-THEN-ELSE statements directly in the code. The difference in computational time between in-line code and a rule-base object was significant but not excessive for this application.

7 CONCLUSIONS

A C++ class called Tripod and a file preprocessor were developed to provide a tool for developing rule-base decision support systems. The approach outlined in this paper was found to provide several advantages over traditional procedural programming for this application:

1. The benefits of a rule-base decision making structure (a Tripod object) and the benefits of procedural functions (the wrapper) can be effectively combined.
2. Multiple small rule-bases can be implemented by the wrapper as an alternative to one large rule-base.
3. The application logic can be modified and expanded without changing the program code, and numerical algorithms can be modified or added without changing the rule-base.
4. The approach utilizes memory efficiently and permits interfacing with third party

programs that often reset environmental variables, dominate memory, and return unique status codes.

Based on the Tripod class, a prototype rule-base model was developed with the capability to interface with remote sensors and external meterological data bases. The model was applied to produce a prototype decision support system for winter highway maintenance in the Intermountain West.

The rule-base for the model was developed by interaction with a panel of domain experts. The model was tested on a small sample of historical data and responded favorably. It has the potential to help an inexperienced person make reasonable decisions consistent with those of an experienced person in similar situations.

ACKNOWLEDGMENTS

Support for this project was provided by the Center for Advanced and Applied Transportation Studies at Utah State University.

REFERENCES

[1] J. Walters and N. R. Nielsen. *Crafting Knowledge-Based Systems.* New York: Wiley, 1988.

[2] E. Turban. *Decision Support and Expert Systems: Management Support Systems*, Second Edition. New York: MacMillan Series in Information Systems, 1990.

[3] H. Adeli. *Knowledge Engineering.* New York: McGraw-Hill, 1990.

[4] Information Builders, Inc., *Level 5 Expert System Software Users Manual.* New York: Information Builders, 1986.

[5] Borland International. *Turbo Prolog User's Guide.* Scotts Valley, CA: Borland International, 1988.

[6] Borland International. *Borland C++ Programmer's Guide.* Scotts Valley, CA: Borland International, 1990.

[7] Robertson Software. *WX-View: A Weather Graphics Terminal.* Geneva, IL: Robertson Software, 1988.

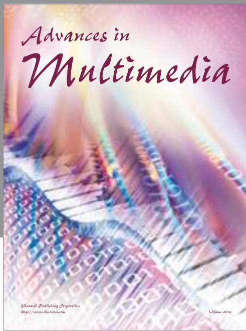
[8] J. D. Bjerregaard. "Use of a weather information system for management of winter highway maintenance." Master of Science Thesis, Utah State University, 1993.

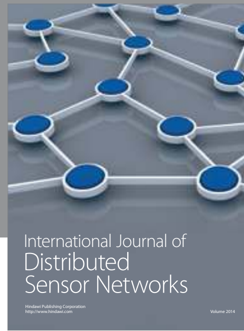
[9] W. J. Grenney and H. N. Marshall. *Artificial Intelligence and Civil Engineering.* Oxford, England: International Conference on the Application of Artificial Intelligence Techniques to Civil and Structural Engineering, 1991, pp. 71–75.

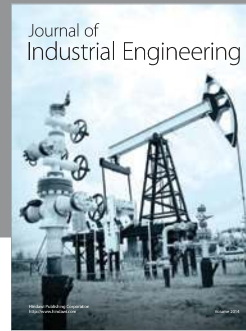
[10] SSI. Equipment manufactured by Surface Systems, Inc, Saint Louis, MO: SSI, 1990.

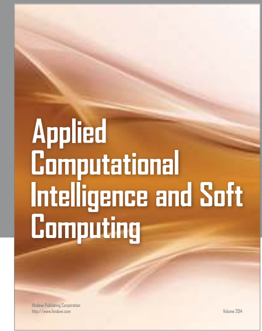
[11] R. K. Penmetsa and W. J. Grenney. "A computer methodology for screening technologies for hazardous waste remediation." *Am. Soc. Civil Eng. Environmental Division*, April 1993 (in press).

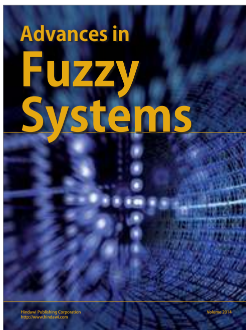
[12] W. A. O'Neill and W. J. Grenney. "Prototype knowledge-base model for matching non-graphic road inventory files with existing digital cartographic databases." Proceedings of the 29th Annual Conference of the Urban and Regional Information Systems Association, San Francisco, CA, 1991.

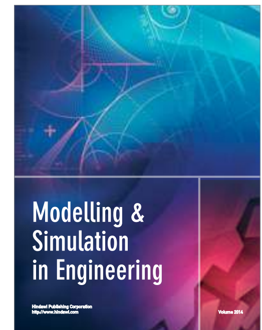












Hindawi

Submit your manuscripts at
http://www.hindawi.com