TrainingNews
Volume IX, Number 3 November 1987

THE CBT CRAFTSMAN

Implementing The Rule-Based Router

Writing the program that matches the *if* with the *then.*

Jesse M. Heines

It is with deepest regret that I report the death of my friend and colleague, Paul Russell, who was the subject of my May, 1987 column "On CBT and Creativity." Paul died peacefully in his sleep due to complications that I am told are common among quadriplegics. Paul had a uniquely cheerful way of looking at the world that was both stimulating and inspiring. I will always cherish the times we spent together discussing the qualities of good CBT in his small room in New Jersey. Paul was without peers in his efforts to produce quality CBT in spite of his handicap.

My last column presented the concepts involved in designing a rule-based router. This column presents the code necessary to implement that router.

To review my last column, rule-based programming is a technique for expressing complex interrelationships in the form of if-then "rules." The "if" part of these rules, typically called the left-hand side (LHS), represents a specific set of circumstances while the "then" part, typically called the right-hand side (RHS), represents the action to be taken when that set of circumstances occurs. To use these rules, the current status of a student (or some other complex system) is expressed in a data item called a "state vector" which is in the same form as a rule LHS. The state vector is then compared to the LHS of each rule in turn until a match is found. The program that does

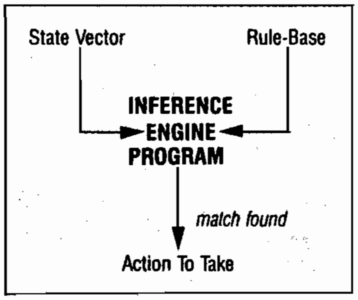

Figure 1. **The state vector is compared to the LHS of each rule in the rule-base. When a match is found, the inference engine program returns the action to take.**

this matching is typically referred to as an "inference engine." When a match is found, the action expressed by the rule's corresponding RHS is taken. The relationships between the state vector, rule-base, and inference engine program are shown in Figure 1.

The problem now is to implement the inference engine program. To maintain consistency with my previous columns, I will show the implementation once again in TenCORE*, an authoring language with powerful programming capabilities. In this language, each line begins with a command word that tells the system what to do. The command word is followed by a command argument that tells the system the parameters needed for that command. The lines are numbered in this column for reference only; TenCORE neither requires nor allows line numbering. All lines that start with an asterisk (*) and all text following double dollar signs ($$) are comments.

As in most programs that work with complex data, we begin by defining the structure of the data. There are two main data structures to consider: the *state vector* and the *rule-base.*

Using the structure defined in my last column, the state vector is an ordered set of five integers: the student's mastery status on each of three skills (with 1 indicating mastery and 0 indicating non-mastery), the number of the module that the student completed last (0-3), and the total number of modules that have been completed (0–5). In TenCORE, we define the state vector as a five-element array of integers called *statevec.* Thus, the status of Skill 1 is stored in *statevec(1),* the status of Skill 2 in *statevec(2),* and so on.

Each rule is also an ordered set of six integers. The first five integers represent a rule's LHS, which is one possible state of the state vector, and the sixth represents the number of the module to be studied next, with 0 indicating that the instructional program should stop. The rule-base defined in my last column had 10 rules, so we need 6 times 10, or 60, integers to store this rule-base. We therefore define the rule-base as a 60-element array of integers called *rules.* (It would be nicer to use a 6 by 10 two-dimensional array to store the rules, but most authoring systems, including TenCORE, only allow one-dimensional arrays if they allow arrays at all. This is not a big problem, because 2D arrays can be mapped onto 1D arrays with an easy algorithm, as you will soon see.)

We want to make the inference engine program as generalized as possible so that it can be used by more than one set of rules. We should therefore implement it as a self-contained subprogram (or unit, in TenCORE terminology). To do this—since TenCORE can only pass parameters that are less than or equal to 8 bytes in length—we must define both *statevec* and *rules* as global variables which are defined for an entire program and can be read or written by any subroutine in that program. These global variables are defined in a special unit called *defines.* While we're at it, let's create a global constant called *nrules* so that in order to use the inference engine program with a different size rule-base, say 12 rules with 4 elements on each LHS, we only need to change the data definitions, not the inference engine program. Unit *defines* is shown in Figure 2.

My last column contained a detailed discussion of how rules are derived and expressed for a sample rule-base developed by Dr. Tim O'Shea. Those rules may be summarized as shown in Figure 3.

To initialize the rule-base for the inference engine program, we need to translate the rules from the form shown above into appropriate values for each of the 60 integers in array *rules.* If we adopt the convention that the asterisk is translated into a value of −1, our job can be made quite simple because all of the non-asterisk values are already integers. The code in Figure 4 will perform the desired assignment.

Likewise, the state vector itself can be initialized to all zeros with the statement in Figure 5.

Since *rules* and *statevec* are global, they do not have to be passed to the inference engine program as parameters; that program can "see" these variables directly. We therefore only need to write that program so that it returns the number of the next module to be studied, *nextmod.* For convenience, however, we will also make the inference engine program return the number of the rule that was applied. If we name

Unit defines

```
1   nrules = 10       $$ number of rules, a global constant
2   rules(60),2       $$ the rule-base, six 2-byte integer entries for each of 10 rules
3   statevec(5),2     $$ the state vector, a list of five 2-byte integers
```

Figure 2. *Statevec* and *rules* must be global variables, defined for the entire program, to be read or written by any subroutine in that program, and defined in a special unit.

Rule Number	LEFT-HAND-SIDE Skill 1	Skill 2	Skill 3	Last Module	No. of Modules	RIGHT-HAND SIDE Next Module
1.	*	*	*	*	5	0
2.	*	*	1	*	*	0
3.	1	*	*	3	*	3
4.	0	*	*	3	*	1
5.	*	1	*	2	*	3
6.	1	*	*	2	*	2
7.	0	*	*	2	*	1
8.	1	*	*	1	*	2
9.	0	*	*	1	*	1
10.	*	*	*	*	*	2

*Notes: a * indicates that the value in this position is irrelevant*
a 0 in the next module column indicates that the instructional program should stop

Figure 3. The rules for a sample rule-base may be summarized this way.

Jesse M. Heines, Ed.D., is an assistant professor of computer science at the University of Lowell in Lowell, Massachusetts. He is the author of **Screen Design Strategies for Computer-Assisted Instruction** *as well as numerous articles on CBT courseware development. Dr. Heines provides training and consultation on computer-based training, develops custom training programs on contract, and writes* **"The CBT Craftsman"** *every other month.*

```
*
* Initialization
*
set      rules(1)  := -1, -1, -1, -1,  5,  0,    $$ rule 1
                      -1, -1,  1, -1, -1,  0,    $$ rule 2
                       1, -1, -1,  3, -1,  2,    $$ rule 3
                       0, -1, -1,  3, -1,  1,    $$ rule 4
                      -1,  1, -1,  2, -1,  3,    $$ rule 5
                       1, -1, -1,  2, -1,  2,    $$ rule 6
                       0, -1, -1,  2, -1,  1,    $$ rule 7
                       1, -1, -1,  1, -1,  2,    $$ rule 8
                       0, -1, -1,  1, -1,  1,    $$ rule 9
                      -1, -1, -1, -1, -1,  2,    $$ rule 10
```

Figure 4. To initialize the rule-base for the inference engine program, we need to translate the rules into the appropriate values for each of the 60 integers in array *rules.*

```
set statevec(1) :=    0,  0,  0,  0,  0,  0,    $$ set all state values to 0
```

Figure 5. The state vector itself can be initialized to all zeros with this statement.

the inference engine program *applyrul*, the TenCORE syntax for calling it would look like this.

```
do      applyrul ( ; rulenum,nextmod )
```

We are now ready to write the inference engine program. The major approach is to begin with the first rule and test the LHS of that rule against the state vector. If that rule's LHS does not match the state vector, we move on and test the LHS of the next rule against the state vector. When a match is found (and if the rules are written correctly, a match will always be found eventually), the inference engine program returns the number of the next module to be studied and the number of the rule whose LHS matched. The code for achieving these results is shown in Figure 6.

The first interesting thing about this code is its brevity. If we were to take out all of the comments (the lines beginning with asterisks), the entire inference engine program would be only 16 lines long! The trick to making the code this concise is in the definition of the rule-base, which explains why I have spent so much time discussing how the rule-base is derived and expressed. But let us now dissect the code to see just how it works.

Lines 8–12 establish three local variables that may be used within this unit without affecting any variables outside of the unit.

Line 16 sets the number of the rule currently being processed to 1 so that the program begins with the first rule in the rule-base when it does its tests.

The nucleus of the program is in lines 21–24 (line 20 is merely a label for the branch in line 30). Line 21 shows the algorithm I referred to earlier for mapping a 2D array onto a 1D array. This line computes an offset into the 60-element rule-base array for the rule currently being tested. This offset is added to all subscript references for the current rule to access the correct elements in the rule-base array. For example, for Rule 2 the offset is 6 * (2 – 1) = 6 * 1 = 6. If we want to refer to the first element of the second rule, we therefore refer to it as *rules(1+6)* which is of course *rules(7).*

Line 22 sets up a loop that is going to execute a maximum of 5 times, once for each element in the state vector (and in the LHS of each rule). Line 23 exits the loop if and only if both of the following conditions are true:

- the current rule element (*rules (k+offset)*) is not equal to –1, i.e., it is not a "wild card," and
- the current rule element is not equal to the corresponding value in the state vector (*statevec(k)*).

A loop exit under these conditions causes an immediate branch to line 25, the statement immediately following the *endloop* command. If either of these conditions is false, control passes to *endloop* command on line 24, at which point the program increments the loop index, *k*, and loops back to line 23 if *k* is less than or equal to 5 or drops through to line 25 if *k* is greater than 5.

Line 25 looks at the value of the loop index to decide whether or not the LHS of the rule being processed matches the state vector. If the loop executed all five times—that is, if at least one condition specified in the *outloop* command was false for each of the five elements in the state vector—the value of *k* at this point will be 6. In this case, the LHS of the rule being processed matches the state vector and the condition in the *branch* command in line 25, *k* is greater than 5, will be true. The program therefore branches to line 34 which contains the label *lmatch.* Line 35 then returns the number of the current rule, *rulenum,* and the number of the next module to be studied, which is the RHS of the current rule and is stored in *rules(6+offset).*

If, on the other hand, the loop terminated prematurely, i.e., if both conditions specified in the *outloop* command were true for one of the five elements in the state vector, the value of *k* at line 25 will be less than or equal to 5. The condition in the *branch* command in line 25, *k* is greater than 5, will be false. The program therefore drops through to line 29 where the number of the rule to be tested is incremented. Line 30 branches the program back to the testing routine to see if the state vector matches this new rule. Note that it is critical that the last rule be a "catch-all" which always matches any state of the state vector. This condition is necessary to prevent an infinite loop.

What we have, then, is a short, relatively simple program that implements the very powerful concept of rule-based routing. Yet although the program is simple, we had to do a great deal of work in setting up the rule-base on which it operates. This somewhat non-intuitive situation, in which deriving the data is more difficult than writing the program that processes it, is common to rule-based programming and other areas of artificial intelligence. While the programming aspect of this work may therefore be handled by junior employees, the derivation of branching data will always remain the task of experienced course developers.

Unit applyrul

```
1  *This unit applies the rules defined in the global variable "rules"
2  * to the state vector stored in the global variable "statevec."
3  * (These must be global variables in TenCORE because they are longer
4  * than 8 bytes and therefore cannot be passed as parameters.) The
5  * unit returns a single integer representing the next teaching operation
6  * to be performed, or -1 if the system should stop.
7  *
8  define local     $$ local variables known only within this unit
9  k,2              $$ loop index
10 offset,2         $$ offset into rule database for current rule
11 rulenum,2        $$ number of rule currently being processed
12 define end
13 *
14 * Initialization
15 *
16 calc      rulenum :=1      $$ the symbol ":=" means "is assigned the value of"
17 *
18 * Test the LHS of a rule for a match with the state vector
19 *
20 1nextrul
21 calc offset := 6 * (rulenum – 1)
22 loop      k := 1,5
23 outloop rules(k+offset) < > –1 $and$ rules(k+offset) < > statevec(k)
24 endloop
25 branch k > 5; 1match; x   $$branch to label 1match if true, drop through if false
26 *
27 * Rule does not match, increment rule counter and apply next rule
28 *
29 calc rulenum := rulenum + 1
30 branch 1nextrul
31 *
32 * A matching rule has been found
33 *
34 1match
35 return rulenum, rules(6+offset)
```

Figure 6. When a match is found, this code ensures that the inference engine program will return the number of the next module and the number of the rule whose LHS matched.

I would be happy to provide TenCORE authors with a demonstration disk containing the source code of the inference engine program; this demo disk also contains a TenCORE unit which calls the inference engine program and displays its results. Please send a formatted diskette and a self-addressed mailer to me in care of *Training News,* 38 Chauncy St., Boston, MA 02111. □

*TenCORE is a registered trademark of Computer Teaching Corporation.