TrainingNews
Volume IX, Number 1 September 1987

THE CBT CRAFTSMAN

The Design Of A Rule-Based Router

Rule-based programming may sound complicated, but it's really just a matter of matching actions to conditions.

Jesse M. Heines

I consider myself basically a CBT implementor, as opposed to a CBT theorist. Theories and concepts are of course necessary, but I find theoretical discussions without complementary implementations hollow, and often downright boring. On the other hand, courses implemented without unifying theories and concepts are at best disjointed and at worst chaotic. My last column ("On Creativity in Programming," July, 1987) presented the concept of creating a menu data structure and using an algorithmic implementation to display the menu and receive the user's selection. That column also showed how this concept could be implemented in a standard CBT authoring language. This month, I want to begin looking at the concept of *rule-based programming*, one of the more basic forms of what is commonly called "artificial intelligence." Rule-based programming is more complex than the programming discussed in my last column, so I will present the concepts this month and leave implementation for my November column.

(Note: I detest the terms "artificial intelligence" and "intelligent CBT" because they have a number of nasty connotations—if my CBT isn't intelligent is it stupid?—and because they have been applied to so many unrelated applications that they are now virtually meaningless buzzwords. Rule-based programming is a distinct technique which I hope to elucidate in this and future columns, and I will therefore use this term exclusively.)

The basic principle in rule-based programming is to express program control structures in the form of *rules*, statements that specify actions to be taken if certain conditions are true. The complete set of rules may be thought of as a *program control data base*, often called a *rule-base*, which is stored in variables and processed algorithmically.

A program that processes a rule-base to extract its inferences is often called an *inference engine*. Part of the problem in developing *practical* rule-based systems is to construct the rule-base so that its rules can be scanned quickly by an efficient inference engine, especially when a large number of rules is involved. Otherwise, a rule-based system can easily become "combinatorially infeasible." Although it may be fine in theory, the amount of processing needed to extract inferences from its rule-base (due to the large number of rule combinations) makes implementation of the system infeasible.

Researchers in rule-based programming have devised a number of forms for expressing and processing rules efficiently for different types of applications. One of the simplest and most common forms is the *if-then* form, in which the left-hand side (LHS) of the rule expresses a number of conditions and the right-hand side (RHS) of the rule expresses an action to be executed if all of the conditions on the LHS are true.

Suppose, for example, that we wish to program a scoring algorithm for a short quiz in which students with scores of 85 or above are classified as masters, those with scores of 60 or below are classified as non-masters, and those with scores of 61 to 84 are presented with a longer, more comprehensive test. In common programming syntax, such rules might look like Figure 1.

```
IF (SCORE >=85) THEN CLASSIFY_STUDENT_AS_A_MASTER
IF (SCORE >=61) AND (SCORE <=84) THEN PRESENT_LONGER_TEST
IF (SCORE <=80) THEN CLASSIFY_STUDENT_AS_A_NON-MASTER
```

Figure 1. One of the simplest and most common types of rule-based programming uses the "if-then" form, in which the left-hand side of the rule expresses a number of conditions and the right-hand side expresses an action to be executed if all the conditions are true.

Writing and processing a series of if-then statements in this form can be burdensome, particularly in the case where more than one LHS might be true. When all of the left-hand sides can be dealt with in terms of a relatively small set of variables, it is usually easier to express the rules in a tabular form, like Figure 2.

Minimum Score	Maximum Score	Student Classification
85	*	1
61	84	2
*	60	3

Notes: * *indicates that the value in this position is irrelevant*
Classification 1 indicates that the student has demonstrated mastery
Classification 2 indicates that no mastery decision can be made
Classification 3 indicates that the student has demonstrated non-mastery

Figure 2. When all of the left-hand sides can be dealt with in terms of a relatively small set of variables, it is usually easier to express the rules in tabular form.

Since this structure is made up completely of numbers, it is easy to implement as a two-dimensional array. The first two elements in each row form the rule's LHS, while the last element forms its RHS. Such a structure can be quickly and easily processed to find which rule matches the student's current score and what classification should be made.

Here's an even more complex example to show how rules are devised. This example, developed by Dr. Tim O'Shea of The Open University in Milton Keynes, England, to illustrate rule-based programming in an instructional setting, is excellent for this type of discussion because it is "bite-sized" and well-documented.

Consider an instructional program in which the terminal objective is to learn a skill referred to as Skill 3. Skill 3 has two sub-skills, Skill 1 and Skill 2, of which Skill 1 is the more basic. Most students should be able to master Skill 2 without going through formal instruction on Skill 1, but we want instruction on Skill 1 to be available for those students who need it. We therefore develop three instructional modules, one for each of the three skills. In addition, we develop three tests to assess student mastery of the skills taught in each of the three modules. The problem is now to decide the order in which these modules should be presented to the student, or how the student should be *routed* through the instructional program to best take advantage of the inherent module prerequisites and the student's ability to master the skills.

The overall plan begins with presentation of Module 2 and then a test on its concepts. Based on the results of this test, the student will be asked to do the same module again or be routed to another module. This module-test-route loop continues until either a maximum of five modules have been presented or the student demonstrates mastery on Skill 3.

We are now ready to begin developing the rule-base for our router. The routing rules will be based on five parameters: the student's mastery status on each of the three skills (with 1 indicating mastery, and 0, non-mastery), the number of the module that the student completed last, and the total number of modules that have been completed. These five parameters form what O'Shea calls a *state vector*, an ordered set of numbers that correspond to the student's

Jesse M. Heines is an assistant professor of computer science at the University of Lowell in Lowell, Massachusetts. He is the author of Screen Design Strategies for Comptuer-Assisted Instruction *as well as numerous articles on 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.*

status on each parameter. For example, a state vector of:

0 1 0 2 1

indicates that the student has demonstrated mastery on Skill 2 but not Skill 1 or Skill 3, that the last module completed was Module 2, and that the total number of modules completed is 1. The state vector defines the format for the rules' left-hand sides. The rules' right-hand sides are simply the number of the module to be studied next, with 0 indicating that the instructional program should stop.

To simplify things, let us put two small constraints on how our rule-base will be processed. First, each rule will be processed sequentially until a rule is found whose LHS matches the state vector. Second, when a matching rule is found, processing of the rules will cease. This constraint eliminates the problem of what to do if the state vector could match the left-hand sides of two or more rules.

The first rules we want to write are those that tell the router when the instructional program should stop. I stated in my description of the overall instructional plan that the module-test-route loop should terminate under two conditions. The first condition occurs when a maximum of five modules have been presented, regardless of the student's mastery states. This condition can be expressed in rule form as shown in Figure 3.

The second terminating condition occurs when the student has demonstrated mastery on Skill 3. In the format shown above, this condition can be expressed in rule form as:

2. * * 1 * * 0

We now consider what we want to happen after the student has completed each module, beginning with Module 3. If Module 3 has just been presented and the student has demonstrated mastery on the corresponding test, Rule 2 will match the state vector and the program will stop. Students who did not demonstrate mastery on the Module 3 test may lack some of the prerequisites for this module. We know that a student must have demonstrated mastery on Module 2 to get to Module 3, so we only need to look at the Module 1 status. If the student has demonstrated mastery on Module 1, we repeat Module 3. Otherwise, we route the student to Module 1. This scheme may be concisely expressed with two rules:

3. 1 * * 3 * 3
4. 0 * * 3 * 1

Rules 2, 3, and 4 take care of all the situations in which Module 3 has just been presented. We now turn to Module 2. If students demonstrate mastery on the Module 2 test, we route them to Module 3. If not, we once again look at their status on Module 1. If they have demonstrated mastery there, we repeat Module 2. Otherwise, we route them to Module 1. This scheme may be concisely expressed with three rules:

5. * 1 * 2 * 3
6. 1 * * 2 * 2
7. 0 * * 2 * 1

Module 1 presents the most basic information, so if students have just completed it and demonstrated mastery on the test, we want them to go onto Module 2. Otherwise, they should simply repeat Module 1:

8. 1 * * 1 * 2
9. 0 * * 1 * 1

Finally, we need a catch-all rule that traps situations in which none of the rules' left-hand sides matches the state vector. This case occurs before the student starts the program, when the state vector is initialized to all zeros. In this case, we want the student to start with Module 2. We therefore write the final rule this way.

10. * * * * * 2

We now have a complete set of rules, one of whose LHS matches each possible value of the state vector. The state vector will be updated by the CBT program each time a student goes through the module-test-route loop.

The next problem is to write a routine that compares the state vector to each of the rules' left-hand sides in turn until a match is found and then returns the value of that rule's right-hand side, which is the number of the module that should be presented next. And this problem will be the subject of my next column. □

Rule Number	LEFT-HAND SIDE					RIGHT-HAND SIDE
	Skill 1	Skill 2	Skill 3	Last Module	No. of Modules	Next Module
1.	*	*	*	*	5	0

*Note: * indicates that the value in this position is irrelevant*

Figure 3. An instructional program's "module-test-route" loop should terminate under two conditions. The first, expressed here in rule form, occurs when a maximum of five modules have been presented, regardless of the student's mastery states.