Ministry of Education and Science of the Russian Federation
Peter the Great St. Petersburg State Polytechnic University
Institute of Computer Sciences and Technologies
Graduate School of Cyber-Physical Systems and Control

Practice Task – Ch 8

Rule Based System
Discipline: Intellectual Computing
3 April 2017

Student Group: 13541/8

Christopher W. Blake

Professor

Kuchmin A.Y

St. Petersburg
2017

Christopher W. Blake

Contents

Christopher W. Blake

Introduction

Chapter 8 of "AI Application Programming" by M. Tim Jones is about the Rule Based Systems. A rule based system uses a working memory and set of rules. The rules are described by two components (antecedents and consequents) which define the triggers and actions to be performed.

A sample C# program has been created to show this type of system. A redundant sensing system is created and the rule based system is used for automatically adjusting it when failures occur. The rules for this system are defined in an external file which is easily adjusted by the user. It should be noted that the actual program runs within the logic of the sample program. Hence, different new rule applications can be created, simply by changing the rule file.

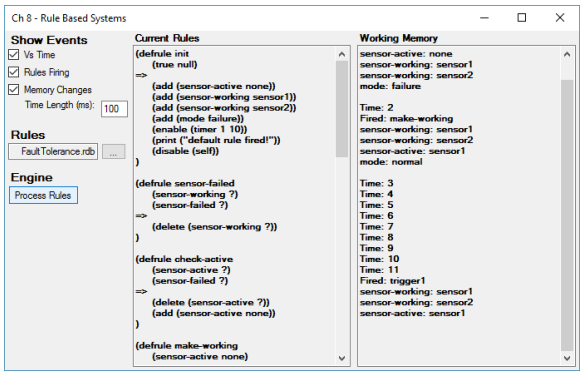

Figure 1: C# Sample Program

Background

Process

The main process flow is explained using the figure to the right (borrowed from the book). This figure depicts the workflow for a rule based system.

1) Initialize system, loading any required initial memory states.
2) Read the rules file
   a) Convert text to antecedents.
   b) Convert text to consequents.
3) Compare antecedents to working memory.
   a) Identify which match working memory.
   b) Obtain first rule that will modify memory.
4) Perform rules of consequent
5) Repeat steps 3 and 4 until no further changes occur.

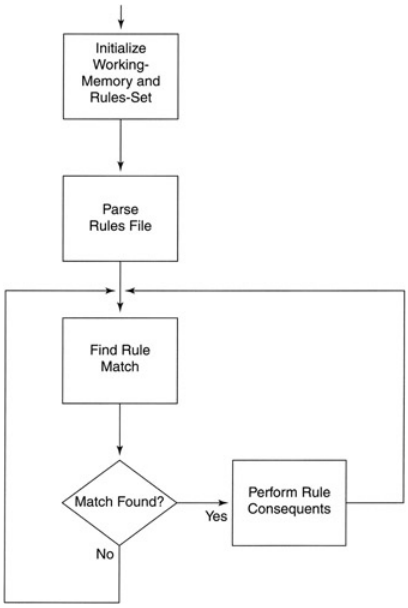

Figure 2: Process Workflow

Rule Definition

A rule has three components: name, antecedents, and consequents. These are all defined in a separate text file (see figure 3 and appendix 1). A wild card character of "?" is used to pass memory values between different antecedents and consequents.

```
(defrule <rule-name>
        (antecedent-terms)
=>
        (consequent-terms)
)
```

Figure 3: Rule definition template

The antecedents are a list of match requirements to working memory. If all antecedents match the memory, then it is possible to trigger the rule. A wild card character of "?" is used between the antecedents as a common variable. It is also provided to the consequent, if required.

The consequents are a list of actions to be performed on the working memory. Items may be added or removed from memory. The wild card character "?" is used like a variable to use a value from the antecedents. Additionally, timers may be used to trigger specified rules by name at a given time.

Representation as Artificial Neural Networks

It is interesting to point out that a rule based system (RBS) is essentially similar to artificial neural networks (ANNs). Both are essentially a system of if-then logic gates. An RBS compares its antecedents to a memory stream, which is analogous to the input nodes of an ANN.

However, the biggest different would be in where and how the knowledge is stored within the system. An ANN must be trained using example data, while an RBS uses predefined rules. In a sense, an ANN could be used to generate the rules that would normally be manually placed into a RBS. However, this is in practice difficult or impossible because the values are stored as very non-human-readable values in a potentially large network of node weights.

Results

The following sensor system is monitored versus time. The results are shown below. This can be compared to the rules (See appendix 1), and it correctly following all situations, simulating the initialization, running, failure, and restarting of two sensors.

```
Time: 1
Fired: init
sensor-active: none
sensor-working: sensor1
sensor-working: sensor2
mode: failure

Time: 2
Fired: make-working
sensor-working: sensor1
sensor-working: sensor2
sensor-active: sensor1
mode: normal

Time: 3,4,5,6,7,8,9,10

Time: 11
Fired: trigger1
sensor-working: sensor1
sensor-working: sensor2
sensor-active: sensor1
mode: normal
sensor-failed: sensor1

Time: 12
Fired: sensor-failed
sensor-working: sensor2
sensor-active: sensor1
mode: normal
sensor-failed: sensor1

Time: 13
Fired: check-active
sensor-working: sensor2
mode: normal
sensor-failed: sensor1
sensor-active: none

Time: 14
Fired: make-working
sensor-working: sensor2
mode: normal
sensor-failed: sensor1
sensor-active: sensor2

Time: 15,16,17,18,19,20

Time: 21
Fired: trigger2
sensor-working: sensor2
mode: normal
sensor-failed: sensor1
sensor-active: sensor2
sensor-failed: sensor2

Time: 22
Fired: sensor-failed
mode: normal
sensor-failed: sensor1
sensor-active: sensor2
sensor-failed: sensor2

Time: 23
Fired: check-active
mode: normal
sensor-failed: sensor1
sensor-failed: sensor2
sensor-active: none

Time: 24
Fired: failure
sensor-failed: sensor1
sensor-failed: sensor2
sensor-active: none
mode: failure

Time: 25,26,27,28,29,30

Time: 31
Fired: trigger3
sensor-failed: sensor2
sensor-active: none
mode: failure
sensor-working: sensor1

Time: 32
Fired: make-working
sensor-failed: sensor2
sensor-working: sensor1
sensor-active: sensor1
mode: normal

Time: 33,34,35,36,37,38,39,40

Time: 41
Fired: trigger4
sensor-working: sensor1
sensor-active: sensor1
mode: normal
sensor-working: sensor2
```

Conclusion

A generic rule based system was created. Using this rule based system, and a specified rule database file, automatic logic can be performed. The example of a redundant sensor system is utilized. This redundant sensor system is initialized, has a failure on one of the sensors, and is automatically returned to working state. Given a rule database file, complex logic can easily be utilized to solve for solutions or control a system.

Christopher W. Blake

Appendix 1 – Example Rule Data Base file

```
(defrule init
      (true null)                    ; antecedent
=>
      (add (sensor-active none))     ; consequents
      (add (sensor-working sensor1))
      (add (sensor-working sensor2))
      (add (mode failure))
      (enable (timer 1 10))
      (print ("default rule fired!"))
      (disable (self))
)

; Define active rule-set
(defrule sensor-failed
      (sensor-working ?)
      (sensor-failed ?)
=>
      (delete (sensor-working ?))
)
(defrule check-active
      (sensor-active ?)
      (sensor-failed ?)
=>
      (delete (sensor-active ?))
      (add (sensor-active none))
)
(defrule make-working
      (sensor-active none)
      (sensor-working ?)
=>
      (add (sensor-active ?))
      (delete (mode failure))
      (add (mode normal))
      (delete (sensor-active none))
)
(defrule failure
      (mode normal)
      (sensor-active none)
      (sensor-failed sensor1)
      (sensor-failed sensor2)
=>
      (add (mode failure))
      (delete (mode safe))
      (delete (mode normal))
)

; Use triggers to simulate timed events...
(defrule trigger1
      (timer-triggered 1)
=>
      (print ("Sensor 1 failure.\n"))
      (add (sensor-failed sensor1))
      (enable (timer 2 10))
      (delete (timer-triggered 1))
)
(defrule trigger2
      (timer-triggered 2)
=>
      (print ("Sensor 2 failure.\n"))
      (add (sensor-failed sensor2))
      (enable (timer 3 10))
      (delete (timer-triggered 2))
)
(defrule trigger3
      (timer-triggered 3)
=>
      (print ("Sensor 1 is now working.\n"))
      (delete (sensor-failed sensor1))
      (add (sensor-working sensor1))
      (enable (timer 4 10))
      (delete (timer-triggered 3))
)
(defrule trigger4
      (timer-triggered 4)
=>
      (print ("Sensor 2 is now working.\n"))
      (delete (sensor-failed sensor2))
      (add (sensor-working sensor2))
      ;(enable (timer 1 10))
           (quit (self))
      (delete (timer-triggered 4))
)
```