

**CSCI3344 - AI**  
**Project #3**  
**Due: Tuesday, April 29, 2008**

**A forward Chaining Rules-Based System**

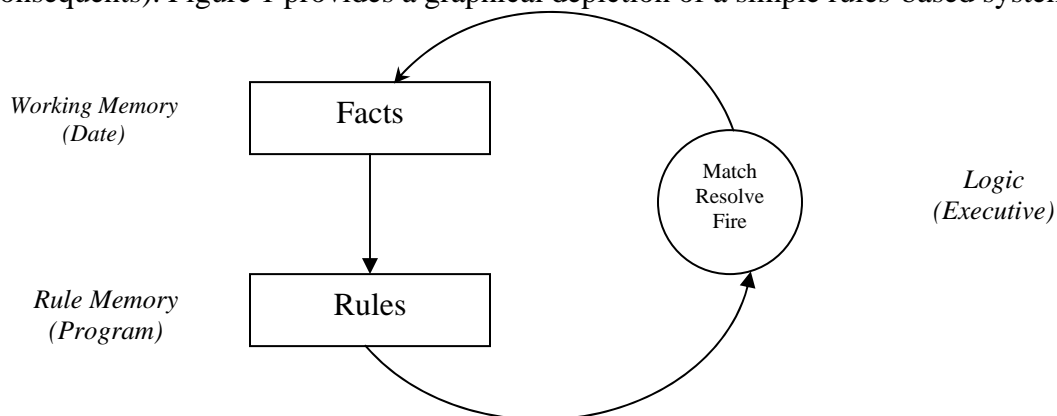
In this project, you will develop a knowledge-based system. These systems are also called production systems, where knowledge is encoded in rules. Knowledge (or facts) is stored in a working memory, and the rules are applied to the knowledge to create more knowledge. This process continues until some goal state is reached. You will investigate a simple rules-based system along with an application in the domain of fault tolerance.

1. **INTRODUCTION**

While a number of different types of rules-based systems exist, we'll focus on a combination of two particular kinds called the deduction system and the reaction system. A deduction system consists of rules representing antecedents and consequents. An antecedent is a condition (an "if" statement, if you will) while the consequent represents the resulting action (the "then" portion). By deduction, the rules insert new facts into the working memory that were "deduced" (reasoned by deduction) from the existing working memory by a given rule. A reaction system includes "actions" that are performed as part of the consequents, such as issuing a command in an embedded system to alter the environment.

2. **ANATOMY OF A RULES-BASED SYSTEM**

A rules-based system is made up of a number of distinct elements. A set of rules exists that operate over a collection of facts stored in a working memory. Logic is provided to identify which rule to fire (based upon the antecedents) and then modify working memory (based upon the consequents). Figure 1 provides a graphical depiction of a simple rules-based system.



**Figure 1** Rules-based system illustration.

Rules operate over the body of facts stored within the working memory. Once a rule is matched, it is permitted to fire, which may (or may not) alter the working memory. This process continues until a goal state is reached.

## Working Memory

The working memory is the structure where the currently known facts are stored. This is a persistent space that can be altered only through the consequent of a rule (and from which a rule may only be fired if its antecedents are true). Consider the following example of working memory:

```
(sensor-failed sensor1)
(mode normal)
```

In this example, two facts are known. These are encoded in pairs of type and value. For example, the first fact is defined as type `sensor-failed` and its value is `sensor1`. The knowledge being demonstrated by this fact is that `sensor1` is failed sensor. The second fact defines that the particular mode is normal. All facts are coded in this representation (though in commercial rules-based systems, a richer representation is often provided).

## Rules Memory

The rules memory contains a set of rules that operate over the working memory. Rules are constructed in two parts and include an “antecedent” and a “consequents.” The antecedent defines the facts that must be true for the rule to be triggered. The consequent defines those actions that will be taken if the rule is triggered. Consider the following example:

```
(defrule sensor-check
  (sensor-failed sensor1)
=>
  (add (disable sensor1))
)
```

The `defrule` token specifies that we’re defining a rule for the system, and is followed by a text name for the rule. This is followed by one or more antecedents (in this case `(sensor-failed sensor1)`) The ‘=>’ symbol separates the antecedents from the consequents. One or more consequents are then provided. These actions are performed only if the rule is triggered. Finally, the rule is closed by the trailing parenthesis. The actions are defined as two elements. The first is the command that is to be issued and the second is a parameter on which the command operates. In this example, if the rule were triggered, the fact `(disable sensor1)` would be added to the working memory.

## Logic

The system of reasoning behind the rules-based system is the logic that identifies the rules to trigger and permits them to fire. This process is commonly defined as a match-resolve-act cycle, and is discussed in section 4 “Phases of a Rules-Based System.”

### **3. TYPES OF RULES-BASED SYSTEMS**

Before we descend into the logic behind the rules-based system, it's important to understand that there are fundamentally two different types of systems. These are forward chaining systems and backward chaining systems.

#### **Backward Chaining**

The backward chaining system is an inference strategy that begins with a hypothesis (a goal state) and works backwards through the rules to generate a new hypothesis and ultimately the currently known set of facts. By arriving at the initial set of facts from the hypothesis, the hypothesis is proven.

#### **Forward Chaining**

The forward chaining system is an inference strategy that begins with known facts. The rules memory is then consulted to identify the rules that match the given set of facts, which may introduce new facts into the working memory. This process continues until either no new facts may be derived, or a goal state is reached. This is a deduction process, which simply uses the known facts to flow through the working memory (and rules) to generate new facts.

This project will focus on forward chaining in both the examples and sample implementation.

### **4. PHASES OF A RULES-BASED SYSTEM**

Let's now look at the phases of a rules-based system, per the forward chaining inference strategy (as shown in Figure 2).

#### **Match Phase**

In the match phase, each of the rules is checked to see if the set of antecedents for the rule can be matched with facts in the working memory. If so, then the rule is added to the conflict set. Rules whose antecedents are not matched are ignored. Once all rules have been checked, the conflict set is then reviewed by the next phase, the conflict resolution phase

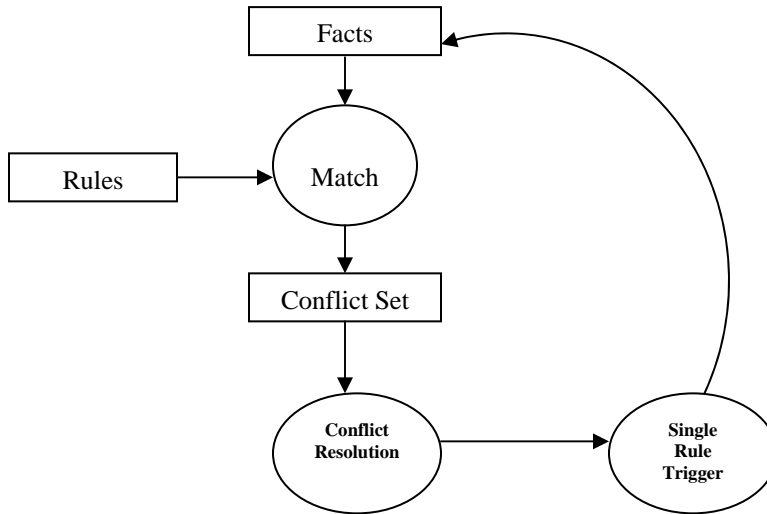


Figure 2 Rules-based system phases.

### Conflict Resolution Phase

The purpose of the conflict resolution phase is to pick a rule to fire out of the conflict set. If only one rule exists in the set, then the process is simple. Given more than one rule in the set, some criteria must be defined to determine which rule to fire. This can be as elaborate as identifying the rule with the greatest number of antecedents (or consequents), or as simple as picking the first rule matched. Upon identifying the rule to be fired, it is passed on to the action phase.

### Action Phase

The action phase performs the set of consequents for the particular rule to be fired. These actions could be adding facts to working memory, removing facts from working memory, or performing some other action. For example, if our rules-based system was connected to the outside world, the actions could manipulate the environment (such as moving a robotic arm or disabling a switch).

## 5. SIMPLE EXAMPLE

Let's now look at a simple example that includes a number of rules and variety of facts in a working memory. You will use a subset of the ZOOKEEPER rules [Winston 1993].

Consider the following rules shown in Listing 1.

**Listing 1** Sample Problem Adapted from Winston's ZOOKEEPER example.

```
(defrule bird-test
  (has-feathers ?)
=>
  (bird ?)
)
(defrule mammal-test
  (gives-milk ?)
)
(defrule ungulate-test1
  (mammal ?)
  (chews-cud ?)
=>
  (is-ungulate ?)
)
(defrule ungulate-test2
  (mammal ?)
  (has-hoofs ?)
=>
  (is-ungulate ?)
)
```

Now consider that our working memory contains the following facts:

```
(gives-milk animal)
(has-hoofs animal)
```

With begin with the match phase where we try to match the facts in our working memory with the antecedents in our rule set. Neither fact matches the antecedents of the first rule (bird-test), but an antecedent is matched in the second test (mammal-test). We save this rule off into our conflict set and continue our search through the remaining rules. No other rules match given the current working memory, so in this case we have the simple case of a conflict set containing one rule. Since there is no conflict, the mammal-test rule is permitted to fire (the action phase), resulting in the working memory as shown below:

```
(gives-milk animal)
(has-hoofs animal)
(mammal animal)
```

We begin again at the match phase and walk through the rules looking for matching antecedents. In this case, our conflict set contains two rules, mammal-test and ungulate-test2. Conflict resolution is also simple given this case, because only one rule here has any affect on the working memory (ungulate-test2). The first rule has previously fired; therefore, nothing new can be added to our working memory and the rule can be omitted from the conflict set. Moving on to the action phase and firing the ungulate-test2 rule results in the following working memory.

(gives-milk animal)

(has-hoofs animal)

(mammal animal)

(ungulate animal)

From this very simple example, our rules system has deduced that the animal in question is an ungulate (a hoofed animal). It reasoned this after first determining that the animal was a mammal (knowing that it gave milk) and then using this information, along with the hoof information, determined that the animal was an ungulate.

Conflict resolution in the example shown included two of the most basic mechanisms for determining which rule to fire. Other mechanisms could be provided for scenarios that are more complicated. For example, the conflict resolver could pick the rule to fire from the conflict set that had the largest number of antecedents. This provides for the most complicated case and could help the system reason to the most relevant goal state and ignore the simpler cases, which might lead the system down unnecessary paths.

## 6. SAMPLE APPLICATION

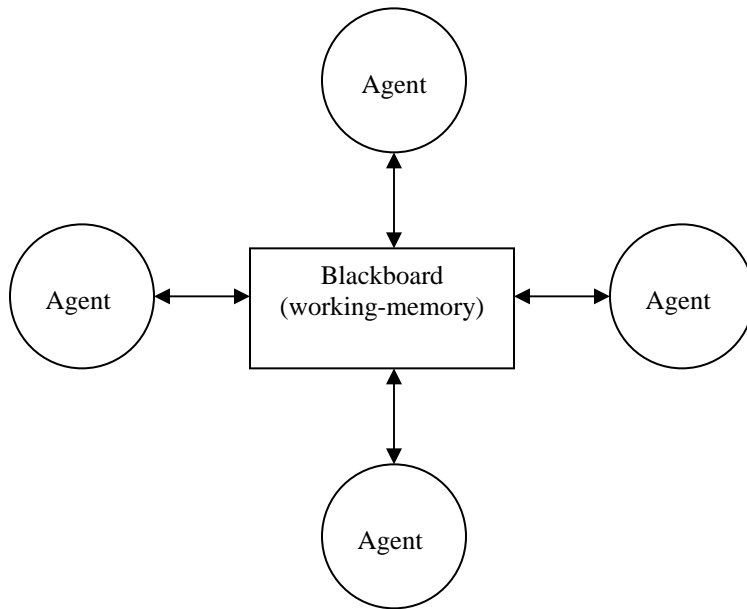
We'll now look at another example that provides a form of fault tolerance within an embedded domain. You are required to use this application to test your system.

### **Fault Tolerance**

In a very simple application, we'll encode rudimentary knowledge of sensor management into our rules. A redundant set of sensors exists from which one can be set as the active sensor (and thus be used by the larger system). Either of the sensors may be working (or failed), but only one may be active at any given time. If neither of the sensors is working, then neither can be active.

In this application, it is assumed that we're operating in a blackboard architecture (see Figure 3). Blackboard architecture contains a number of agents that use and produce data on the blackboard. The blackboard serves as a working space where facts are communicated amongst the agents. Agents are triggered by data on the blackboard and can manipulate the data (adding, changing, or removing), which may trigger other agents to continue the work process.

The blackboard architecture is interesting because it not only provides the ability for agents to communicate with one another, but also to coordinate and synchronize their activities.



**Figure 3** Graphical depiction of a blackboard architecture.

## Rules Definition

The rules file for our sample fault tolerance demonstration is shown in Listing 2. This set of rules introduces a number of new consequents (actions) that can be performed.

### Listing 2 Fault Tolerance Demonstration Rules 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))
  (delete (timer-triggered 4))
)

```

This example contains nine rules, but only four are actual operating rules. The first rule (`init`) in Listing 2 is an initialization rule. Note that the antecedent is `(true null)`, which always resolves to true. This initialization rule permits us to seed the working memory with an initial set of facts (as defined by the `add` commands). The `enable` command allows us to enable a timer, which when fired can trigger another rule. We'll use timers in this example to perform simulated events from the environment. These would be events (facts) placed into the working memory from other agents operating within the environment (recall Figure 3). The `timer` command uses two arguments, the numeric id of the timer and the number of seconds until the timer should fire (in this case, timer 1 will fire in 10 seconds). The `print` command is a debugging command that provides us with some visibility into the reasoning of the system. The final command in this rule, `disable`, is very important. It provides the ability to disable the rule from the set of available rules. This rule can then never be fired again, an obvious need for rules whose antecedents are always matched (such as the `init` rule).

Once the `init` rule has fired, the working memory will contain:

```

(sensor-active none)
(sensor-working sensor1)
(sensor-working sensor2)
(mode failure)

```

Rule `make-working` would be the next rule to fire (using `sensor1` as the matched parameter), leaving working memory as:

```

(sensor-working sensor1)
(sensor-working sensor2)
(sensor-active sensor1)
(mode normal)

```

At this point, no further rules can fire and the system remains in this state until the timer fires (after 10 seconds). When a timer fires, the system simply adds the fact to working memory (`(timer-triggered 1)`). This new fact in working memory then causes the rule to fire that handles the particular timer (using the `trigger events` as the rule's antecedent). The event triggered by the timer simulates the failure of `sensor1`, modifying working memory as follows:

```

(sensor-working sensor1)

```

```
(sensor-working sensor2)
(sensor-active sensor1)
(mode normal)
(sensor-failed sensor1)
```

Rules sensor-failed and check-active then fire, leaving working memory in the following state:

```
(sensor-working sensor2)
(mode normal)
(sensor-failed sensor1)
(sensor-active none)
```

Finally, rule make-working fires, leaving working memory as:

```
(sensor-working sensor2)
(mode normal)
(sensor-failed sensor1)
(sensor-active sensor2)
```

At this point, the system is operational with a working sensor from the redundant pair. The rules also provide for no working sensors with an indication of this event using the mode fact.

Name the fault-tolerance script fault.rbs.

## **REFERENCE**

[Winston 1993] Winston, P. H. 1993. Artificial Intelligence. 3<sup>rd</sup> edition. Addison Wesley.

### **What you have to turn in**

Your system should be able to provide a command line interface for the user. Under the command line prompt, the user expects to be able to load fault.rbs and execute it. For example,

```
$java rbs.Main
Rbs>(batch fault.rbs)
Rbs>(run)
```

where rbs is the supposed name of your system. Note that this example also assumes you program in java. But this is not required. You can program in any programming language you like.

The two main things to turn in are your source code (program code and fault.rbs) and a written report (single space, font 12) that presents

- A brief overview of the problem.
- Experiment design.

You do not have to use Listing 1 as the content of fault.rbs. Feel free to design your own sensor fault tolerance rules based on the idea and format of Listing 1, and describe your design. **20 extra credits** will give to the design showing innovation.

- Test results.