National Research Council Canada | Conseil national de recherches Canada

Institute for Information Technology | Institut de technologie de l'information

NRC-CNRC

Fuzzy Reasoning in Jess: The FuzzyJ Toolkit and FuzzyJess

Orchard, R.
July 2001

* Proceedings of the ICEIS 2001, Third International Conference on Enterprise Information Systems, Setubal, Portugal. July 7-10, 2001. pp. 533-542. NRC 44882.

Copyright 2001 by
National Research Council of Canada

Permission is granted to quote short excerpts and to reproduce figures and tables from this report, provided that the source of such material is fully acknowledged.

Canada

FUZZY REASONING IN JESS: THE FUZZYJ TOOLKIT AND FUZZYJESS

Robert Orchard
National Research Council of Canada, Institute for Information Technology,
Montreal Road, Building M-50, Ottawa, Ontario, Canada K1A 0R6
Email: bob.orchard@nrc.ca

Key words: fuzzy logic, Jess, fuzzy reasoning, java, fuzzy toolkit, expert system, FuzzyJ, FuzzyJess, FuzzyCLIPS

Abstract: Jess, the Java™ Expert System Shell, provides a rich and flexible environment for creating rule-based systems. Since it is written in Java it provides platform portability, extensibility and easy integration with other Java code or applications. The rules of Jess allow one to build systems that reason about knowledge that is expressed as facts. However, these facts and rules cannot capture any uncertainty or imprecision that may be present in the domain that is being modelled. This paper describes an extension to Jess that allows some forms of uncertainty to be captured and represented using fuzzy sets and fuzzy reasoning. We describe the NRC FuzzyJ Toolkit, a Java API that allows one to express fuzzy concepts using fuzzy variables, fuzzy values and fuzzy rules. Next, we describe a Java API called FuzzyJess that integrates the FuzzyJ Toolkit and Jess. Finally, we show the modifications that were made to the Jess code to allow this extension (and others with similar requirements) to be added with modest effort and with minimal or no impact as new releases of Jess are delivered.

1. INTRODUCTION

This paper describes an extension to Jess (Friedman-Hill 2001) that provides a fuzzy reasoning capability. The extension, called FuzzyJess, is a Java™ [1] API that allows one to use the FuzzyJ Toolkit (Orchard 2001) with Jess to define fuzzy concepts and to create fuzzy rules using these concepts. It would be beneficial for the reader to have some knowledge of fuzzy reasoning or to review some introductory information or books (Kosko 1997; Cox 1994; Tsoukalas 1997) that deal with the topic. We begin with a brief overview of the FuzzyJ Toolkit showing how to create a simple fuzzy system using Java code.

Then we describe the public FuzzyJess interface and illustrate, using the same example, the corresponding Jess program for this fuzzy system. Finally we describe the hooks that were provided in the Jess code and the private FuzzyJess code that allowed the extension to be built with a modest effort and in such a way that future revisions of Jess should have little or no impact on FuzzyJess.

[1] Java and all Java based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

2. FUZZYJ TOOLKIT

The FuzzyJ Toolkit provides a capability for modelling fuzzy concepts and reasoning in a Java setting. Much of the work is based on earlier experience with the FuzzyCLIPS (Orchard 1998) extension to the CLIPS Expert System Shell (Riley 2001).

Fuzzy concepts are represented using fuzzy variables, fuzzy sets and fuzzy values. A **FuzzyVariable** is used to describe a general fuzzy concept. It consists of a name (for example, air temperature), its units (such as Degrees C), a range

(for example, from 0 to 100), and a set of fuzzy terms that can be used to describe specific fuzzy concepts for this variable. The fuzzy terms are defined using a term name such as *cold* or *hot*, together with a **FuzzySet** that identifies the degree of membership of the term over the range of the fuzzy variable. In **Figure 1** we show the fuzzy set representations of the terms *cold*, *OK*, and *hot* for a fuzzy variable *air temperature*. Notice that the term *cold* has a high degree of membership at 0 degrees C, since we are certain that a temperature of 0 degrees C is cold. The degree of membership drops linearly as the temperature moves toward 20 degrees C, indicating that our certainty that the temperature is cold is decreasing. After 20 degrees C it has a value of 0, meaning it is definitely not cold.

The fuzzy variable terms along with a set of system or user supplied fuzzy modifiers (like *very* or *slightly*), the operators *and*, *or* and *not* and the left and right parentheses provide the basis for a simple grammar that allows one to write **fuzzy linguistic expressions** to describe fuzzy concepts in an english-like manner. These expressions are encoded in a **FuzzyValue** representing a specific fuzzy concept. For example, the expression

air temperature is very cold or hot

is composed of the terms *cold* and *hot*, along with the fuzzy modifier *very* and the operator *or*.

The logic of expert systems is often encoded in rules. In the FuzzyJ Toolkit these are fuzzy rules. A **FuzzyRule** holds three sets of FuzzyValues representing the *antecedents*, *conclusions* and *input* values of the rule. A rule might be written as follows:

if a_1 and a_2 … and a_n
then c_1 and c_2 … and c_m

The antecedents (a_i) are the premises of the rule that must be *true* before the rule can execute (fire) and the conclusions (c_i) of the rule can be asserted. In non-fuzzy systems like Jess, the antecedents and conclusions are **crisp** and for an antecedent to be true the facts (knowledge) of the system must match exactly with the antecedent. Consider the crisp rule:

```
If   air temperature is 29.5 degrees C
then set fan speed to 711.0 RPM
```

In this case the temperature must be exactly 29.5 degrees C before the rule will fire. If the temperature is 29.501 degrees C, the rule will not fire. Now consider a fuzzy variation of this rule:

```
If   air temperature is hot
then set fan speed to high
```

In this case the air temperature needs only to match the fuzzy concept of hot to some degree for the antecedent to be *true* and for the rule to fire. It will assert a fuzzy value for fan speed that takes this into account so that the actual fan speed will vary according to the degree of hotness of the current temperature. This single rule represents a large number of discrete crisp rules and is much closer to the way we would naturally express such a concept.

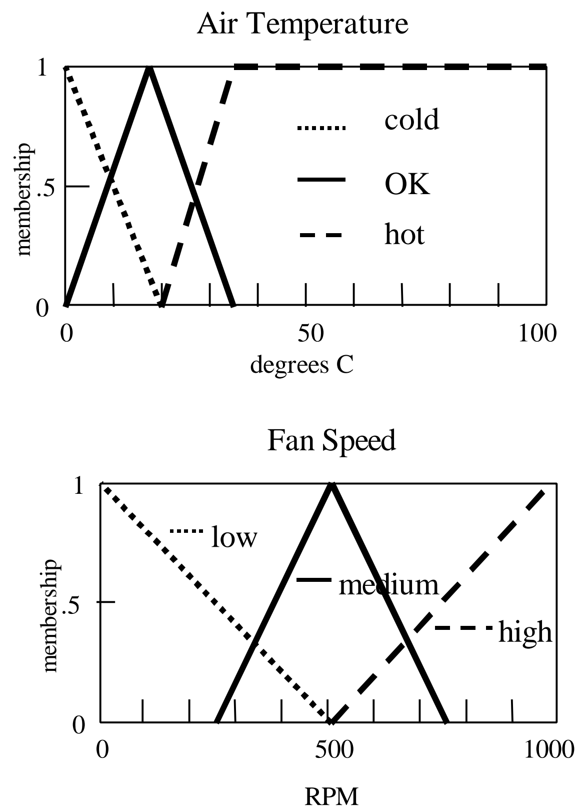

Figure 1

Note that with the FuzzyJ Toolkit we can create simple rules that have only fuzzy values on the left hand side (antecedents) and right hand side (conclusions). Many fuzzy systems can be created using the toolkit with Java code, but as we will see in the next section, when integrated with Jess it provides the ability to create more complex rules, combining crisp and fuzzy values.

To demonstrate this we will use a simple example with two fuzzy variables, air temperature and fan speed. For air temperature we define the terms *cold*, *OK* and *hot*. For fan speed we define the terms *low*, *medium* and *high*. The terms are represented graphically in fuzzy sets as shown in **Figure 1**. Notice how the concepts overlap. For example, an air temperature of 5 degrees C is

considered to be both cold and OK to some degree (more cold than OK). With these definitions we create three rules (**Table 1**) that allow us to set the fan speed depending on the current air temperature.

What follows is a Java program that implements this simple fuzzy system, demonstrating the creation and use of FuzzyVariable, FuzzyValue, FuzzySet and FuzzyRule objects. Details about the methods and arguments used can be found in the API section of the FuzzyJ Toolkit User's Guide (Orchard 2001).

Table 1

Rule
if air temperature is cold **then** set fan speed to low
if air temperature is OK **then** set fan speed to medium
if air temperature is hot **then** set fan speed to high

```
import nrc.fuzzy.*; // the FuzzyJ Toolkit classes
public class Demo1
{ static public void main(String args[])
  { FuzzyRule coldTemp = new FuzzyRule(); FuzzyRule OKTemp = new FuzzyRule();
    FuzzyRule hotTemp = new FuzzyRule(); FuzzyValueVector fanSpeedfvv = null;
    FuzzyValue globalFanSpeedfv = null, currentTempfv = null;
    FuzzyVariable airTemp = null, fanSpeed = null;
    RightLinearFunction rlf = new RightLinearFunction();
    LeftLinearFunction llf = new LeftLinearFunction();
    try
    { // define the fuzzy variables for air temperature and fan speed
      // with their associated linguistic terms
      airTemp = new FuzzyVariable("airTemperature", 0.0, 100.0, "Deg C");
      airTemp.addTerm("cold", new RFuzzySet(0.0, 20.0, rlf));
      airTemp.addTerm("OK", new TriangleFuzzySet(0.0, 20.0, 35.0));
      airTemp.addTerm("hot", new LFuzzySet(20.0, 35.0, llf));
      fanSpeed = new FuzzyVariable("fanSpeed", 0.0, 1000.0, "RPM");
      fanSpeed.addTerm("low", new RFuzzySet(0.0, 500.0, rlf));
      fanSpeed.addTerm("medium", new TriangleFuzzySet(250.0, 500.0, 750.0));
      fanSpeed.addTerm("high", new LFuzzySet(500.0, 1000.0, llf));

      // define the 3 fuzzy rules each with a single antecedent and
      // a single conclusion fuzzy value
      coldTemp.addAntecedent(new FuzzyValue(airTemp,"cold"));
      coldTemp.addConclusion(new FuzzyValue(fanSpeed,"low"));
      OKTemp.addAntecedent(new FuzzyValue(airTemp,"OK"));
      OKTemp.addConclusion(new FuzzyValue(fanSpeed,"medium"));
      hotTemp.addAntecedent(new FuzzyValue(airTemp,"hot"));
      hotTemp.addConclusion(new FuzzyValue(fanSpeed,"high"));
    }
    catch (FuzzyException fe)
    { System.err.println("Error initializing fuzzy variables/rules\n" + fe);
      System.exit(100);
    }
    // iterate over a range of temperatures determining the fan speed that
    // should be selected for that temperature
    for (double t = 0.0; t <= 40.0; t = t + 2.0)
    { // since each rule that fires can generate a fuzzy value for the
      // fan speed we need to assimilate all of the fuzzy value outputs
      // into a single global fuzzy value (globalFanSpeedfv)
      globalFanSpeedfv = null;
      // clear inputs to rules on each iteration so we can set the new ones
```

```
      coldTemp.removeAllInputs(); OKTemp.removeAllInputs();
      hotTemp.removeAllInputs();
      try
      { // add inputs to rules; note that we convert the crisp temperature
        // to a fuzzy value - this process is known as fuzzification
        currentTempfv = new FuzzyValue(airTemp, new TriangleFuzzySet(t,t,t));
        coldTemp.addInput(currentTempfv); OKTemp.addInput(currentTempfv);
        hotTemp.addInput(currentTempfv);
        // execute the 3 rules and determine fan speed in global FuzzyValue
        // globalFanSpeedfv; for each rule test to see if rule input(s) match
        // antecedent(s) and if so execute rule; if rule fires add resultant fan
        // speed fuzzy value to global fan speed by performing a fuzzy union
        if (coldTemp.testRuleMatching())
        { fanSpeedfvv = coldTemp.execute();
          if (globalFanSpeedfv == null)
            globalFanSpeedfv = fanSpeedfvv.fuzzyValueAt(0);
          else
            globalFanSpeedfv =
                   globalFanSpeedfv.fuzzyUnion(fanSpeedfvv.fuzzyValueAt(0));
        }
        if (OKTemp.testRuleMatching())
        { fanSpeedfvv = OKTemp.execute();
          if (globalFanSpeedfv == null)
            globalFanSpeedfv = fanSpeedfvv.fuzzyValueAt(0);
          else
           globalFanSpeedfv =
                   globalFanSpeedfv.fuzzyUnion(fanSpeedfvv.fuzzyValueAt(0));
        }
        if (hotTemp.testRuleMatching())
        { fanSpeedfvv = hotTemp.execute();
          if (globalFanSpeedfv == null)
            globalFanSpeedfv = fanSpeedfvv.fuzzyValueAt(0);
         else
            globalFanSpeedfv =
                   globalFanSpeedfv.fuzzyUnion(fanSpeedfvv.fuzzyValueAt(0));
        }
        // output the result for the given air temperature - the fan speed
        // fuzzy value is defuzzified to give a crisp result
        System.out.println("For temp = " + t + " Fan speed set to " +
                            globalFanSpeedfv.momentDefuzzify() + " RPM");
      }
      catch (FuzzyException fe)
      { System.err.println(fe); System.exit(100);}
    }
  }
}
```

Some of the output of this FuzzyJ program is shown below.

```
For temp = 18.0 Fan speed set to 464.6499238964992 RPM
For temp = 20.0 Fan speed set to 500.0 RPM
For temp = 22.0 Fan speed set to 546.354852876592 RPM
```

The program is complete and simple enough that it should be obvious how fuzzy variables and fuzzy rules are defined. A couple of comments should clarify the less obvious parts of the code. First the input temperature is crisp. In order to use the temperature as an input to the rules, it needs to be in the form of a fuzzy value. A common way to do this is to define a fuzzy set that has the shape of a triangle with no width, in fact defining full membership at the crisp value and no membership everywhere else. That is what was done in this example. Another option is to make the triangle narrow and centered about the crisp value. This might represent the idea of imprecision in the device that measures the temperature. Changing a crisp value to a fuzzy value is often referred to as *fuzzification*. Another thing that needs some explanation is the idea that for a given input, multiple rules can fire and each rule that fires makes a contribution towards the total or global solution. In this example, if the air temperature is 10 degrees C, it is equally cold and OK. This means that two rules will fire, one suggesting that the fan speed be set to low and the other that it be set to medium. Each rule contributes to the final solution and the two outputs are combined (using a union of fuzzy values) to give a global fuzzy value. This process is sometimes called *global contribution*. Finally this global fuzzy value output must be converted back to a crisp value to set the fan speed. This process is called *defuzzification*. There are many ways to do this. The FuzzyJ Toolkit provides two such methods: the average of maximums over the fuzzy set and the moment or center of gravity of the fuzzy set.

3. FUZZYJESS

This section describes the classes, methods and user functions that make up FuzzyJess. We show an implementation, using Jess code, of the same fuzzy system defined in the previous section.

From the Jess language perspective there is very little added. In fact there is an implementation of the Jess Userpackage interface, the FuzzyFunctions class, that adds just three fuzzy functions to Jess. These functions are *fuzzy-match*, *fuzzy-rule-similarity* and *fuzzy-rule-match-score*. The latter two are beyond the scope of this paper (and covered in the FuzzyJ Toolkit User's Guide [2]). The fuzzy-match function takes two arguments: either both FuzzyValue objects or a FuzzyValue object and a string that represents a valid fuzzy expression. If one of the arguments is a string then it will be converted to a FuzzyValue using the FuzzyVariable associated with the other FuzzyValue argument. The FuzzyVariable objects must both be associated with the same FuzzyVariable so that they can be compared (i.e. we can't compare air temperature and fan speed). If there is some degree of match[2] between the two, then the fuzzy-match function returns true, otherwise it returns false. Below we have the Jess equivalent of the example presented in the previous section. Note the use of the fuzzy-match function in the patterns of three of the rules. For those who have seen and/or used FuzzyCLIPS, it appears to be a little more complex. The patterns in FuzzyCLIPS would be written as

```
  (airTemp cold)
rather than
  (airTemp ?t&:(fuzzy-match ?t "cold"))
```

However, FuzzyJess does provide a great deal more flexibility in the fuzzy patterns and does not require internal changes to any Jess parsing routines.

Also, when fuzzy facts are asserted in the rules, FuzzyJess automatically takes care of the *global contribution* issue. As identical fuzzy facts are asserted from different rules the contribution from each rule is accumulated. In the previous section when using the FuzzyJ Toolkit alone, we had to take care of this with quite specific Java coding. Observe that each rule that contributes to the same fuzzy output does fire independently. Because of this it is necessary to allow all of these rules to fire before the final global conclusion is used. This is done using the salience property in the *control* rule. By setting it to a value of –100 it will fire only after the other rules have fired. Some have argued that this global assimilation of results should be done automatically by the expert system shell, however, it is not possible to identify the rules that will generate the same fuzzy outputs and have them be bumped in priority to execute together. This is due to the flexibility of systems like CLIPS and Jess. One can't determine the outputs without executing the rules since the results can be buried in variable values and function calls. One system, FuzzyShell [8], does make this claim, but it appears to do so at the expense of flexibility (all outputs must be determinable at compile time).

[2] The level of this degree of matching can be set. By default it will be any match greater than 0.0. See the static FuzzyValue method, in the FuzzyJ Toolkit API documentation (Orchard 2001)].

```
(import nrc.fuzzy.*)
;; Two globals to hold our FuzzyVariables for air temperature and fan speed
(defglobal ?*airTempFvar* = new FuzzyVariable "airTemperature" 0.0 100.0 "Deg C"))
(defglobal ?*fanSpeedFvar* = (new FuzzyVariable "fanSpeed" 0.0 1000.0 "RPM"))

(defrule init "An initialization rule that adds the terms to the FuzzyVariables"
  =>
   ;; the nrc FuzzyJess functions are loaded
   (load-package nrc.fuzzy.jess.FuzzyFunctions)
   (bind ?rlf (new RightLinearFunction)) (bind ?llf (new LeftLinearFunction))
   ;; terms for the air temperature Fuzzy Variable and fan speed Fuzzy Variable
   (?*airTempFvar* addTerm "cold" (new RFuzzySet 0.0 20.0 ?rlf))
   (?*airTempFvar* addTerm "OK" (new TriangleFuzzySet 0.0 20.0 35.0))
   (?*airTempFvar* addTerm "hot" (new LFuzzySet 20.0 35.0 ?llf))
   (?*fanSpeedFvar* addTerm "low" (new RFuzzySet 0.0 500.0 ?rlf))
   (?*fanSpeedFvar* addTerm "medium" (new TriangleFuzzySet 250.0 500.0 750.0))
   (?*fanSpeedFvar* addTerm "high" (new LFuzzySet 500.0 1000.0 ?llf))
   ;; assert the first the fuzzy input -- temperature at 0.0
   (assert (crispAirTemp 0.0))
   (assert (airTemp (new FuzzyValue ?*airTempFvar* (new TriangleFuzzySet 0.0 0.0 0.0)))))

(defrule temp-cold-fanSpeed-low "if air temperature cold then set fan speed low"
   (airTemp ?t&:(fuzzy-match ?t "cold"))
 =>
   (assert (fanSpeed (new FuzzyValue ?*fanSpeedFvar* "low"))))

(defrule temp-OK-fanSpeed-medium "if air temperature OK then set fan speed medium"
   (airTemp ?t&:(fuzzy-match ?t "OK"))
 =>
   (assert (fanSpeed (new FuzzyValue ?*fanSpeedFvar* "medium"))))

(defrule temp-hot-fanSpeed-high "if air temperature hot then set fan speed high"
   (airTemp ?t&:(fuzzy-match ?t "hot"))
 =>
   (assert (fanSpeed (new FuzzyValue ?*fanSpeedFvar* "high"))))

(defrule control "printing of results and initiating next iteration"
   ;; to combine output of all 3 rules we must wait until the 3 rules
   ;; have all fired ... low salience for this rule achieves this
   (declare (salience -100))
   ?catf <- (crispAirTemp ?t)
   ?fsf <- (fanSpeed ?fuzzyFanSpeed)
 =>
   ;; defuzzify the fan speed fuzzy value and print out the result
   (bind ?crispFanSpeed (?fuzzyFanSpeed momentDefuzzify)) (bind ?t (+ ?t 2.0))
   (printout t "For temp = " ?t " Fan speed set to "?crispFanSpeed " RPM" crlf)
   (if (<= ?t 50.0) then
       (retract ?catf ?fsf) (assert (crispAirTemp ?t))
       (assert  (airTemp (new FuzzyValue ?*airTempFvar* (new TriangleFuzzySet ?t ?t ?t))))))
```

Some of the output of this FuzzyJess system is shown below (identical to the output from the Java version).

```
For temp = 18.0 Fan speed set to 464.6499238964992 RPM
For temp = 20.0 Fan speed set to 500.0 RPM
For temp = 22.0 Fan speed set to 546.354852876592 RPM
```

There are a couple of other things that a user must know to use the FuzzyJess extension. You need to have access to the FuzzyJ Toolkit and FuzzyJess packages (nrc.fuzzy and nrc.fuzzy.jess). Normally these will be in a Java jar file for easy inclusion in the *classpath* variable. The only other thing that is required is that instead of using a **Rete** object in programs, you must use a **FuzzyRete** object. For convenience the classes nrc.fuzzy.jess.FuzzyConsole and nrc.fuzzy.jess.FuzzyMain have been provided and they can simply replace any use of jess.Console or jess.Main. Consider the code for FuzzyMain:

```
public class FuzzyMain extends Main
{public static void main(String[] argv)
  {FuzzyMain m = new FuzzyMain();
   m.initialize(argv, new FuzzyRete());
   m.execute(true);
  }
}
```

and the code for FuzzyConsole:

```
public class FuzzyConsole extends Console
{public FuzzyConsole(String name)
  {super(name, new FuzzyRete());
  }
 public static void main(String[] argv)
  {new FuzzyConsole("Fuzzy Jess Console")
                 .execute(argv);
  }
}
```

4. JESS MODIFICATIONS

The previous two sections have dealt with the FuzzyJ Toolkit and FuzzyJess from the perspective of a user who wants to build fuzzy systems. In this section we will discuss the requirements for building extensions like FuzzyJess, in particular what was done to the Jess code to enable a reasonably simple implementation and some of the details of the FuzzyJess classes that were developed.

A fuzzy rule fires in Jess when the fuzzy (and crisp) patterns on the left hand side of the rule match. The fuzzy matching is controlled by the use of the fuzzy-match function. However, when the right hand side of the rule is executed, it is often necessary to know what fuzzy values matched the fuzzy patterns specified in the fuzzy-match function calls. In particular, this information is required when a fuzzy fact is being asserted since the *shape* of the fuzzy value being asserted depends on the degree of matching of the fuzzy patterns on the right hand side. To accommodate this a number of things were done.

First, since we need to know whether a fact was being asserted in a rule activation or outside of any rule activation, the Jess Rete class was modified so that it calls the method, *aboutToFire*, whenever an rule begins to fire and another method, *justFired*, when it has completed. In standard Jess these methods are empty (do nothing). In FuzzyJess we extend the Rete class with our own **FuzzyRete** class and override these two methods so that the current Activation object is recorded when the rule is about to fire and cleared when the rule has finished execution. This allows us to know when the right hand side of a rule is being executed and provides access to needed information in the Activation object during the rule firing.

Next it was necessary to perform special processing of fuzzy facts being asserted. A rule might perform the following assert on it's right hand side:

```
(assert (fanSpeed (new FuzzyValue
                 ?*fanSpeedFvar* "low")))
```

But the actual shape of the fuzzy value in the asserted fact depends on the degree of matching of any fuzzy values on the left hand side of the rule. To this end, the Jess Rete class has a method, doPreAssertionProcessing, that is called whenever a fact is being asserted. The FuzzyRete class overrides this method to do special processing, modifying any fuzzy values as necessary and then calling the parent class *doPreAssertionProcessing* method to do the standard Jess processing.

To actually calculate the required shapes for the fuzzy values in asserted facts (and for execution of the two Fuzzy Jess user functions, *fuzzy-rule-similarity* and *fuzzy-rule-match-score*) we need to remember the fuzzy value pairs that matched on the left hand side of the rule. Without providing any detail about the Jess implementation (pattern net and join net, etc.), we can say that when a set of facts satisfy the patterns and tests of the left hand side of a rule an Activation object is created. The Activation object identifies the rule it is associated with and also the facts that matched the patterns and passed the tests. These facts are referenced in a list of Token objects. If we had the simple (crisp) rule

```
(defrule simple
      (A)
      (B)
   =>
      (assert (C)))
```

and we asserted the facts (A) and (B), an Activation object would be created that identified the *simple* Rule object and two Token objects (linked together), one referencing the (A) fact and the other referencing the (B) fact. These tokens were just the place where we could store the fuzzy value pattern and the fuzzy value input pairs when a fuzzy-match is successful. So we extended the Token class, creating a new **FuzzyToken** class to store the extra information. Since each pattern (or Test) could result in multiple calls to fuzzy-match, we store the matching fuzzy pattern/fuzzy input pairs in a vector, m_extensionData. When the fuzzy-match function is successful, it creates a FuzzyValueVector with the two fuzzy values and then stores this in the private m_extensionData vector of the FuzzyToken. Let's look at the following fuzzy rule.

```
(defrule fuzzy-person
  (person (name ?name)
     (height ?ht&:(fuzzy-match ?ht "tall"))
     (weight ?wt&:(fuzzy-match ?wt "heavy"))
   )
  =>
   (assert (coat (new FuzzyValue
                  ?*coatSizeFvar* "large"))))
```

If we assert the fact

```
(person (name john)
  (height (new FuzzyValue ?*height*
           "very tall"))
  (weight (new FuzzyValue ?*weight*
           "slightly heavy")))
```

then fuzzy-match will execute twice during the matching of the fact with the pattern and we will store two FuzzyValueVectors in the m_extensionData vector of the FuzzyToken. One of the fuzzy pattern/fuzzy input pairs is tall/very tall and the other is heavy/slightly heavy. Actually this doesn't quite do the job by itself. If one of the patterns is matched and the other fails to match the overall matching of the fact will fail, yet there will be a pair stored in m_extensionData. This needs to be cleared. Also Tokens are shared as they propagate through the join network. In a crisp environment this is not a problem since the Token can be shared by multiple activations. However, since the same fuzzy fact might match different patterns in different rules and since we need to store the fuzzy pattern/fuzzy input pairs so they are available during activation of these rules, it is not appropriate that they be shared. This led to a slightly more extensive change being made to the Jess code. In appropriate places, after all tests are done (via calls to functions such as fuzzy-match) in a pattern match, a call to a new method, *prepare*, is made to allow the extension to deal with the success or failure of tests. In the case of the fuzzy extension, we set the m_extensionData field to null if the tests fail and duplicate the FuzzyToken if the tests are successful (and there was a call to fuzzy-match). At this point the correct information is available for the right hand side of a rule when it fires.

The proper assertion of fuzzy facts is handled in the doPreAssertionProcessing method of the FuzzyRete object. This is called whenever an assertion is being done. If the fact is fuzzy (i.e. has fuzzy values in any of its slots), the fuzzy match pairs for the activation are obtained (remember from above that we stored the Activation object in the FuzzyRete object when the rule began to fire so we could access this information). These pairs together with the fuzzy values for the asserted fact are used to create a FuzzyRule[3]. The fuzzy patterns are the rule's antecedents, the fuzzy inputs are it's inputs and the fuzzy values in the fact's slots are the consequents for the rule. The FuzzyRule is executed and the output fuzzy values are used to replace the values in the fact's slots. At this point we have the required fuzzy fact for assertion according to this rule in isolation. Finally a check is made to see if there is an existing fuzzy fact that is identical[4] to the one about to be asserted. In standard Jess with only crisp facts, what is done depends on the setting of the fact duplication flag. However, with fuzzy facts there is no duplication and the new fuzzy fact is combined[5] with an existing identical fuzzy fact (global contribution) to create a new fact and the old fact is retracted.

Although users do not need to understand all of these implementation details, it will help if they have an understanding of the general behavior of the system. Consider the following two similar looking rules and the suggested fuzzy fact assertion.

```
(defrule r1-good
    (temp ?t&:(fuzzy-match ?t "not hot"))
  =>
    (assert (set-valve (new FuzzyValue
             ?*valve-change* "positive"))))
```

[3] Note that this FuzzyRule is only created once and is reused for each fuzzy fact that is asserted during that firing.

[4] Two fuzzy facts are identical if they have identical crisp values and the fuzzy values in the same fact position have the same FuzzyVariable.

[5] In the current implementation the fuzzy values are combined by doing a union of their fuzzy sets.

```
(defrule r2-not-so-good
    (temp ?t&:(not (fuzzy-match ?t "hot")))
  =>
    (assert (set-valve (new FuzzyValue
             ?*valve-change* "positive"))))

(assert (temp (new FuzzyValue ?*temperature*
                "cold")))
```

The first rule, *r1-good*, will fire since *cold* and *not hot* will overlap (match to some degree) and the fuzzy-match function will be successful. It will store the *cold* and *not hot* fuzzy values and use these to determine how to modify the output fuzzy value *positive* to reflect the degree of matching. The output will **not** be the value *positive*. The second rule, *r2-not-so-good*, will also fire. The fuzzy-match function will fail since *cold* and *hot* do not overlap, but the *not* operation will cause the test to be true. Since the fuzzy-match failed, it does not store the *cold* and *hot* fuzzy values, the fuzzy value of the output fact is not modified and the output will be *positive*. A naïve user might have expected the two rules to give the same result.

5. CONCLUSIONS

With some small changes to Jess, the extension of the Rete and Token classes to FuzzyRete and FuzzyToken classes, the inclusion of the FuzzyFunction class to implement a few new functions (including fuzzy-match) and the addition of a FuzzyFactoryImpl[6] class to allow FuzzyTokens to be created in place of standard Tokens, users can easily build fuzzy systems. There is very little impact on the performance of Jess and as Jess evolves it is expected that FuzzyJess will remain intact, requiring little, if any, modification. Other extensions to Jess may benefit from the additions made to accommodate FuzzyJess, however, adding multiple independent extensions coincidently may require some reworking of the changes since clearly it is not possible to extend the Rete and Token classes more than once without some difficulties.

For simple systems and for performance purposes creating fuzzy systems using only the FuzzyJ Toolkit may be appropriate. But as systems grow larger and the number and type (fuzzy, crisp, fuzzy-crisp) of rules make the system more complex, FuzzyJess may have an advantage since it could be more robust and likely will be easier to maintain. Certainly hybrid approaches of Java and FuzzyJess will also be used in many applications.

This work has provided a useful tool for building certain types of fuzzy systems. We have been looking at various enhancements including the addition of certainty factors (as found in FuzzyCLIPS), support for fuzzy numbers and other forms of inferencing. The actual direction will mostly be determined by feedback from users and the time available to work on the project. We welcome and encourage users to report significant applications of the tools so we may understand the real world uses and gather documentation to support our continued effort.

[6] We don't discuss this here; it allows the FuzzyRete object to specify a *factory* to be used to create tokens. Jess calls Rete.getFactory().newToken(…) to create the appropriate type of token. The factory sets the stage for adding other such extensions.

ACKNOWLEDGEMENTS

The author would like to extend a special thank you to Ernest Friedman-Hill of Sandia National Labs for his contribution to this effort, particularly in cleaning up and streamlining the Jess modifications.

REFERENCES

Friedman-Hill, Ernest J. (2001). *Jess, The Java Expert System Shell* [online]. Sandia National Laboratories. Available from: http://herzberg.ca.sandia.gov/jess [Accessed 19 Apr 2001]

Orchard, R.A. (2001). *NRC FuzzyJ Toolkit for the Java™ Platform User's Guide* [online]. National Research Council of Canada. Available from: http://ai.iit.nrc.ca/IR_public/fuzzy/fuzzyJDocs/index.html [Accessed 19 Apr 2001]

Kosko, Bart, 1997. *Fuzzy Engineering*. Prentice Hall, Inc.

Cox, Earl. 1994. *The Fuzzy Systems Handbook*. Academic Press, Inc.

Tsoukalas, Leferi, and Uhrig, Robert, 1997. *Fuzzy and Neural Approaches in Engineering*. John Wiley & Sons, Inc.

Orchard, R.A., 1998. *FuzzyCLIPS Version 6.04A User's Guide*, ERB-1054, National Research Council of Canada

Riley, Gary. (2001). *CLIPS, A Tool for Building Expert Systems* [online]. GHG Corporation. Available from: http://www.ghg.net/clips/CLIPS.html [Accessed 19 Apr 2001]

Pan J., DeSouza G. N., and Kak A. C., 1998. FuzzyShell: A Large-Scale Expert System Shell Using Fuzzy Logic for Uncertainty Reasoning. *IEEE Transactions on Fuzzy Systems*, 6(4), 563-581