FuzzyShell User's Manual

Juiyao Pan
Robot Vision Laboratory
1285 E.E. Building
Purdue University
West Lafayette, IN 47907-1285
http://rvl1.ecn.purdue.edu/

Feb. 19, 1996

1 What is FuzzyShell ?

FuzzyShell [1] from **Purdue Robot Vision Lab** is a fuzzy expert system shell for large-scale general-purpose applications. This project is motivated by the observation that all the currently available (either through commercial channels or as free-ware over the internet) fuzzy expert system shells work correctly for only one-shot inference. In other words, when rules are allowed to trigger other rules, these system violate some basic tenets of fuzzy logic. We must hasten to add that, as opposed to shells, there do exist many domain-specific fuzzy expert systems that carry out a correct implementation of fuzzy logic. However, besides being domain specific, these systems tend to be small, often involving only tens of rules, as opposed to hundreds or even thousands. The main reason why the commercially available and other fuzzy expert system shells fail to work correctly for multi-step inference are laid out in our paper to which this manual is an accompaniment. Our fuzzy expert system shell, called FuzzyShell, has been designed by generalizing Rete networks – the soul of traditional non-fuzzy large-scale expert systems – to the case of fuzzy logic. Explicitly or implicitly, all well-known shells (e.g., CLIPS, OPS5, OPS83, ART, etc.) are founded on the notion of Rete Networks. A Rete Network basically ameliorates the computational complexity that would otherwise be associated if all the rule-antecedents had to be matched with all the facts for every cycle of the inference engine in order to determine what rules to fire.

FuzzyShell, is an extension of CLIPS5.1. [2] As documented here, the modifications made to CLIPS5.1 are extensive. Unlike CLIPS, FuzzyShell allows the rules and facts to be represented by fuzzy terms characterized by membership functions. The inference engine of FuzzyShell performs

[1]This project is supported by Office of Naval Research.
[2]CLIPS, originally developed by NASA, stands for C Language for Integrated Production System.

fuzzy pattern matching between the rule-antecedents and the facts, and allows fuzzy evidence to be aggregated when fuzzy inferences are made about the same linguistic variable by different rules. To be specific, in FuzzyShell a fact like (robot (speed fast)) will match a rule-antecedent like (robot (speed slow)) if there exists an overlap between the membership functions associated with the terms *fast* and *slow*. FuzzyShell will also correctly fire all the relevant rules and aggregate all the evidence about a linguistic variable before allowing that linguistic variable to trigger further rules in chained inference. Except for the newly created commands that are related to fuzzy reasoning, the syntax of FuzzyShell commands is the same as for CLIPS5.1. This we have done is for the convenience of the user. (However, it hides the fact that the internal workings of the two systems are radically different.)

Like CLIPS, FuzzyShell is designed to be highly portable. FuzzyShell is completely written in C Language and can be run under SGI machines and Sun machines using Sun OS or Solaris. With slight modification, FuzzyShell can also be run on PC and Mac machines. FuzzyShell also provides some useful graphical tools to display intermediate results, such as the membership functions of fuzzy terms and defuzzification results; this is to facilitate program development and troubleshooting.

This manual is organized as follows: First, we talk about how to install FuzzyShell for Sun and SGI machines. Next, an example will be used to quickly illustrate how FuzzyShell is used. The user can use this example as a starting point to gain a deeper understanding of how FuzzyShell is used. That will be followed by showing how to define membership functions for the fuzzy terms associated with linguistic variables. Subsequently, the evidence propagation process through rules and the defuzzification process will be explained. To gain a thorough understanding of FuzzyShell, the user would also need access to the manuals for CLIPS5.1. CLIPS5.1 can be obtained through COSMIC. [3]

2 How to install ?

The user can acquire the executables for FuzzyShell via the following site on World Wide Web(WWW):

$$http://RVL1.ecn.purdue.edu/$$

The source code is also available upon request and with prior arrangement with the Robot Vision Lab. At the WWW site mentioned, the user should click on the item "A Large Scale Fuzzy Expert System Shell" for downloading the executables and other information related to FuzzyShell.

At the www site, the information is organized in the form of the following three files:

- architecture.ps: the document for the implementation of FuzzyShell architecture

[3] CLIPS is available through COSMIC, the NASA software distribution center. COSMIC can be contacted at: COSMIC, 382 E. Broad St., Athens, GA 30602, (404) 542-3265.

- manual.ps: the document for using FuzzyShell
- FuzzyShell.tar.z: the executable FuzzyShell in compressed tar'd format

The two files, architecture.ps and manual.ps, are in postscript format and FuzzyShell is a binary file. The source code for FuzzyShell is composed of the following files:[4]

analysis.c*	build.c*	commline.c*	defrule.c*	deftempl.c*
deftmcom.c*	deftmfun.c*	deftmlhs.c*	deftmpsr.c*	drive.c*
engine.c*	expressn.c*	factcom.c*	factmngr.c*	generate.c*
match.c*	parsutil.c*	reorder.c*	ruleprsr.c*	sysdep.c*
variable.c*	xclips.c*	xmain.c*	xmenu.c*	xmenu_wind.c*
defrule.h*	engine.h*	expressn.h*	facts.h*	network.h*
reorder.h*	setup.h*	symbol.h*	bload.c	bsave.c
classcom.c	classfun.c	conscomp.c	constrct.c	deffacts.c
deffnctn.c	defglobl.c	defins.c	drulebin.c	edbasic.c
edmain.c	edmisc.c	edstruct.c	edterm.c	evaluatn.c
genrcbin.c	genrccmp.c	genrccom.c	genrcfun.c	inscom.c
insfun.c	insquery.c	intrfile.c	lgcldpnd.c	math.c
memory.c	msgcom.c	msgfun.c	multivar.c	objbin.c
objcmp.c	reteutil.c	retract.c	router.c	rulecom.c
scanner.c	spclform.c	strings.c	sysio.c	syspred.c
sysprime.c	sysecnd.c	textpro.c	utility.c	xclipstext.c
xedit.c	analysis.h	bload.h	bsave.h	classcom.h
classfun.h	clips.h	clips.hlp	clipsmem.h	cmptblty.h
colors.h	commline.h	conscomp.h	constant.h	constrct.h
deffacts.h	default.h	deffnctn.h	defglobl.h	defins.h
deftempl.h	deftmfun.h	deftmlhs.h	deftmpsr.h	drive.h
evaluatn.h	ed.h	extobj.h	generate.h	genrcbin.h
genrccmp.h	genrccom.h	genrcfun.h	inscom.h	insfun.h
insquery.h	intrfile.h	lgcldpnd.h	match.h	msgcom.h
msgfun.h	multivar.h	objbin.h	objcmp.h	object.h
parsutil.h	reteutil.h	retract.h	router.h	ruleprsr.h
scanner.h	shrtlnkn.h	spclform.h	sysdep.h	sysprime.h
utility.h	variable.h	xclips.h	xedit.h	xmain.h
xmenu.h	xmenu_wind.h	xsetup.h	deffuzzy.h**	fzfacts.c**
fzutil.c**	fzmatch.c**	fznetwk.c**	fzmod.c**	fzdefztn.c**
fzparser.c**	deffuzzy.h**	fzfacts.h**	fzutil.h**	fzmatch.h**
fznetwk.h**	fzmod.h**	fzdefztn.h**	fzparser.h**	

In what follows, we will describe how to make the executable file of FuzzyShell for Sun or SGI machines. The user is referred to CLIPS5.1 manual for installation on PC and Mac type machines. For SGI and Sun, in a shell window the following commands need to be executed

[4] The files marked by * are CLIPS files with significant modification for FuzzyShell. The files marked by ** are the newly created for FuzzyShell and are not from CLIPS. The rest of files are the same as those of CLIPS.

sequentially as listed.

```
>xmkmf
>make
```

3 How to get started ?

The usual steps to writing your own rule-based expert system in FuzzyShell will now be summarized. Note that for troubleshooting a program, the user always has the option of using the provided utilities for seeing the intermediate results that are generated during the running of the inference engine.

1. Define the membership functions of fuzzy terms for the linguistic variables in your systems. Prepare your own rules. Define the templates and the initial facts if needed. All information above can be stored in a same file or in separate files.

2. Start the FuzzyShell program by typing *FuzzyShell*.

3. Load the file(s) above into FuzzyShell by using the command *load*.

4. Start the inference engine by using the command *run*. The user can specify the number of rules , say (run 1) for one rule at a time, so that the user can use the utilities to see the intermediate results.

5. For some reason, if your own rule-based program is not running well as you expect, try to modify your rules or the membership functions, and then follow the same steps above again.

We will now use a two-rule example for illustration. Note that, in practice, hundreds, even thousands, of rules may be used in FuzzyShell. Besides, any one rule may consist of multiple condition elements that could be far more complex than what we have shown in this example.

Assume there is a file "example.fz" containing the following information.

Example 1

```
(deflingvar speed                              ;;; membership functions for
    (slow 0.0 1.0 10.0 1.0 20.0 0.0)           ;;; linguistic variable speed
    (medium 10.0 0.0 20.0 1.0 40.0 0.0)
    (fast 20.0 0.0 40.0 1.0 50.0 0.0))

(deflingvar object-distance                    ;;; membership functions for
    (close 0.0 1.0 3.0 1.0 5.0 0.0)            ;;; linguistic variable object-distance
    (near 2.0 0.0 5.0 1.0 8.0 0.0)
```

```
    (far 6.0 0.0 10.0 1.0 13.0 0.0))

(deftemplate robot                    ;;; defining the robot object class
    (field mission (type SYMBOL))
    (field object-distance (type SYMBOL)))

(deffacts init-fact                   ;;; some initial facts
    (robot (mission navigation)(object-distance near)))

(defrule rule1-a
    (robot (mission navigation)(object-distance far))
  ⇒
    (assert (speed fast)))

(defrule rule1-b
    (robot (mission navigation)(object-distance near))
  ⇒
    (assert (speed medium)))
```

Shown above, we have defined the membership functions of the fuzzy terms for the linguistic variables *speed* and *distance*. We have also defined rule1-a, rule1-b and the template of *robot*. The user can issue the command *reset* to assert the initial facts. In this case, the fact (robot (mission navigation)(object-distance near)) will be asserted into the working memory. By matching the facts with the rule-antecedents, the inference engine will declare that rule1-a and rule1-b are firable and will place these two rules in the agenda. The user can start running the inference engine by issuing the command *run*. Although the membership functions, the rules and the facts can be defined by loading them from data files into the system, the user always can perform the same task by giving those commands from the command prompt shown by "command>". We will describe how to use the command *deflingvar* to define the membership functions and how to define the facts. The user is referred to CLIPS5.1 manual for detailed explanations of how to use the commands *deftemplate*, *deffacts* and *defrule*.

FuzzyShell provides convenient utilities under X window for the user to diagnose their problem during the system development. Fig.1 is a snapshot of what the monitor may look like during a typical FuzzyShell session. Note particularly the windows displayed for scrutinizing the intermediate results during the inference process. By examining the information displayed in these windows, the user can clearly see how the fuzzy evidence was propagated through the rules, the membership function of the aggregate_term for a linguistic variable, the latest form of the membership functions of the fuzzy terms look reasoned about so far, etc. Later we will discuss in more detail how these windows can be used to facilitate system development.

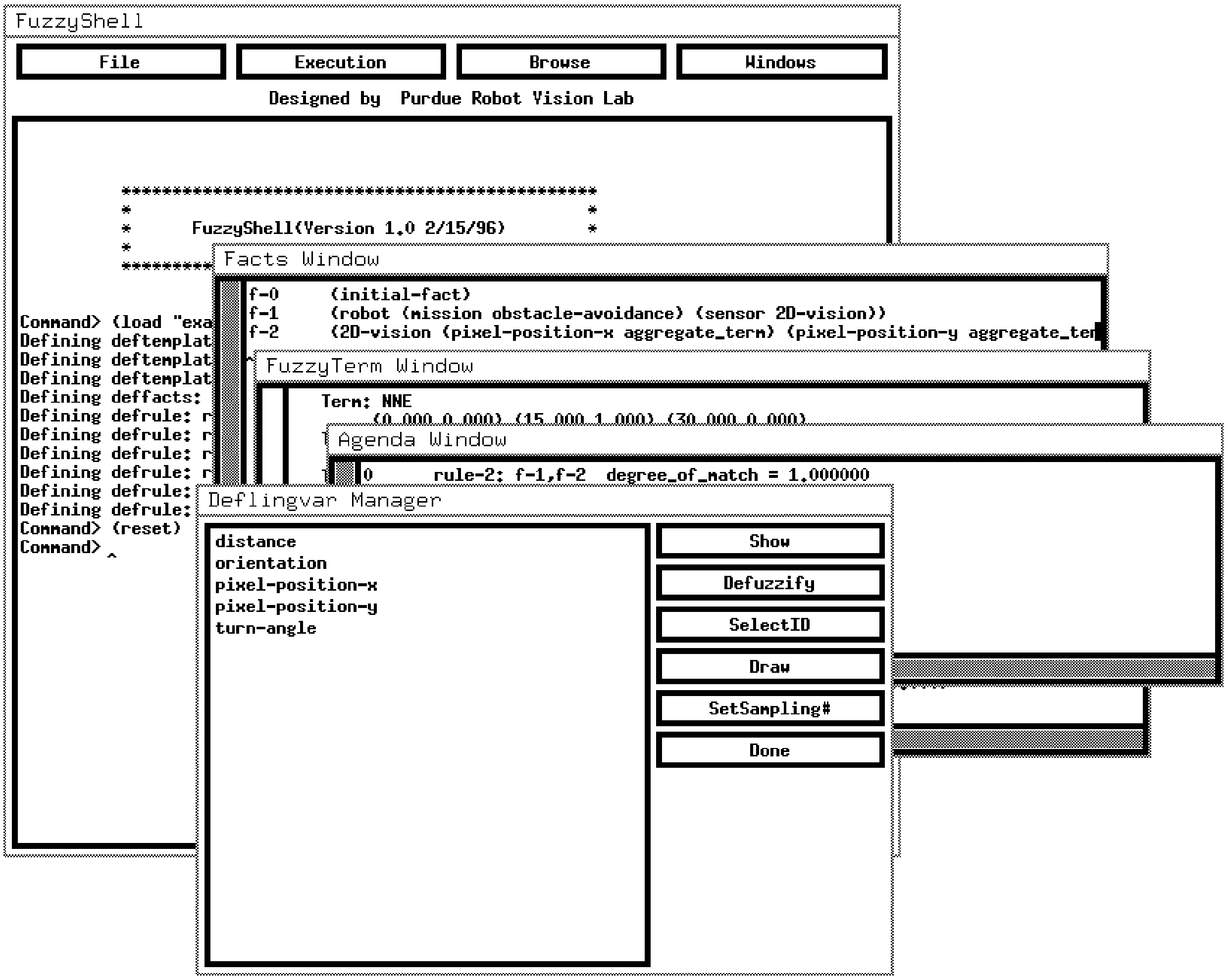

Figure 1: *Screen dump for a typical session with FuzzyShell. The FuzzyShell window is the top-level window for typing in commands. In Deflingvar window, the user can display both the original membership functions of the fuzzy terms and their latest forms.*

4 Creating Linguistic Variables

Membership functions are assumed to be piecewise linear functions in FuzzyShell. The command *deflingvar* defines a linguistic variable, its fuzzy terms and the associated membership functions.

Syntax of deflingvar:

(deflingvar linguistic-variable
    (fuzzy-term-1 x11 d11 x12 d12 .... x1p d1p)
    (fuzzy-term-2 x21 d21 x22 d22 .... x2q d2q)
      ..
    (fuzzy-term-m xm1 dm1 xm2 dm2 .... xmr dmr))

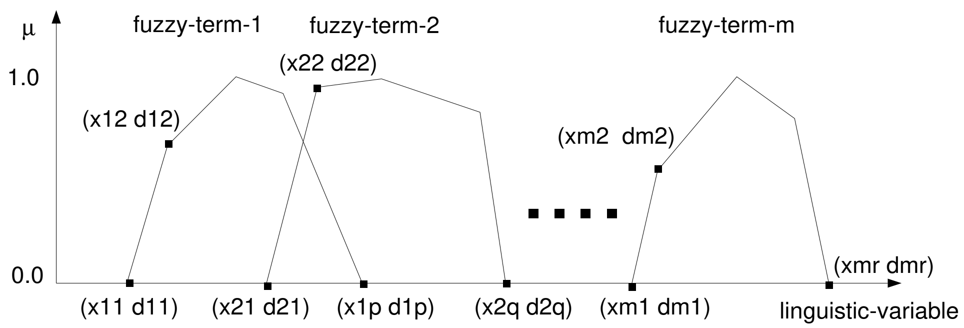

Figure 2: *Shown here is the representation of membership functions for the fuzzy terms in FuzzyShell. This representation allows the membership functions to be piecewise linear.*

Fig.2 shows the form of fuzzy terms associated with a linguistic variable that could be created using the command *deflingvar*. As shown, each membership function is represented by a sequence of points. Each point consists of two values (one for the value in the universe of discourse and the other for the membership grade). For example, fuzzy-term-1 is represented by p points, the first point consisting of the pair (x11, d11), and so on. Any number of fuzzy terms may be defined for a linguistic variable.

Example 2

(deflingvar brake
    (hard 10.0 0.0 14.0 1.0 18.0 0.0)
    (very-hard 15.0 0.0 19.0 1.0 23.0 1.0))

(deflingvar pixel-position
    (very-left 0.0 1.0 50.0 1.0 80.0 0.0)
    (medium-left 40.0 0.0 120.0 1.0 160.0 0.0)
    (somewhat-left 140.0 0.0 200.0 1.0 250.0 0.0)

(center 200.0 0.0 255.0 1.0 310.0 0.0)
(somewhat-right 260.0 0.0 310.0 1.0 370.0 0.0)
(medium-right 350.0 0.0 390.0 1.0 470.0 0.0)
(very-right 430.0 0.0 460.0 1.0 511.0 1.0))

(deflingvar turn-angle
(left-50 -70.0 1.0 -50.0 1.0 -38.0 0.0)
(left-30 -42.0 0.0 -30.0 1.0 -15.0 0.0)
(left-10 -25.0 0.0 -10.0 1.0 0.0 0.0)
(zero -10.0 0.0 0.0 1.0 10.0 0.0)
(right-10 0.0 0.0 10.0 1.0 25.0 0.0)
(right-30 15.0 0.0 30.0 1.0 42.0 0.0)
(right-50 38.0 0.0 50.0 1.0 70.0 1.0))

Example 2 shows the use of *deflingvar* for declaring the linguistic variables *brake*, *pixel-position* and *turn-angle*, their terms and the membership functions for each of the terms. Once these fuzzy terms are defined and loaded into FuzzyShell, the user can use the *Show* button to see their membership functions graphically. Fig.3 shows the selected item, *brake*, and the resulting display of the membership functions for the two terms, *hard* and *very-hard*, for this linguistic variable.

To summarize, to create a linguistic variable and to subsequently examine its terms and the associated membership functions, the user need to select the *deflingvar* under the *Browse* menu at the right top of the main window. A small window will then come up with the names of the defined linguistic variables, with a list of buttons on the right. The user will need to click on the linguistic variable desired and then click on the *Show* button. The user can also pop up the *FuzzyTerm* window to see the actual numerical values for a membership function. The FuzzyTerm window can be popped up by selecting *FuzzyTerm* under the *Windows* item.

With the definitions for *speed*, *object-distance*, *pixel-position* and *turn-angle* in Example 1 and 2, the membership functions of their fuzzy terms are also shown in Fig.4, Fig.5, Fig.6 and Fig.7 using the *Show* utility; these displays will be of help in some of the explanations to follow. In order to discard the window showing the membership function, the user types "q" inside that window.

It is also possible to use the command *show-lingvar* at the top-level for displaying the membership functions of a linguistic variable. For example,

command> (show-lingvar brake)

in the FuzzyShell window will return the following information:

Linguistic Variable: brake
Term: hard 10.0 0.0 14.0 1.0 18.0 0.0
Term: very-hard 15.0 0.0 19.0 1.0 23.0 1.0

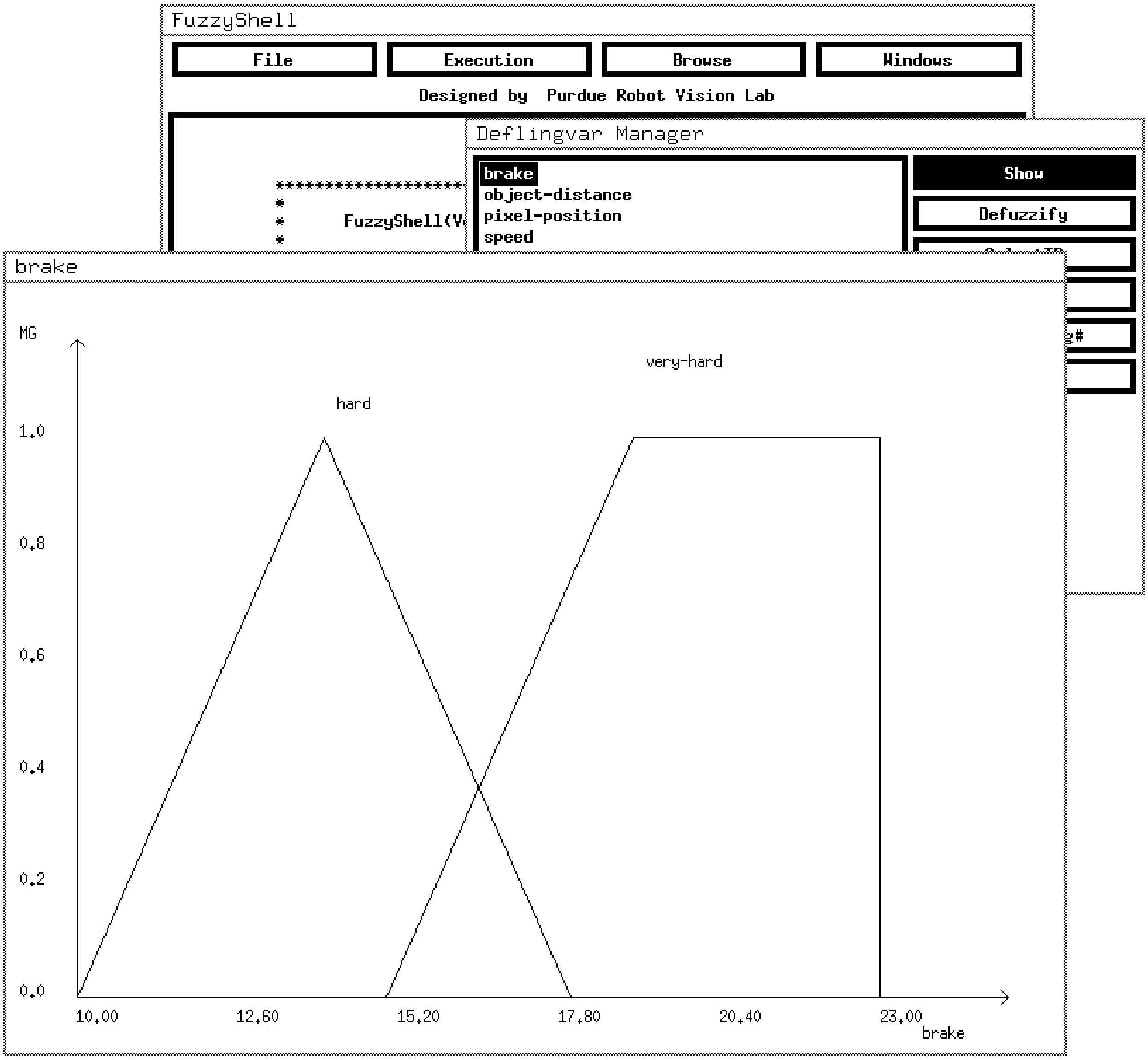

Figure 3: *To show the membership functions of the fuzzy terms for linguistic variable brake, the user needs to pop up the window for Deflingvar Manager under the Browse menu. The linguistic variable desired is then clicked, followed by a click on the Show button.*

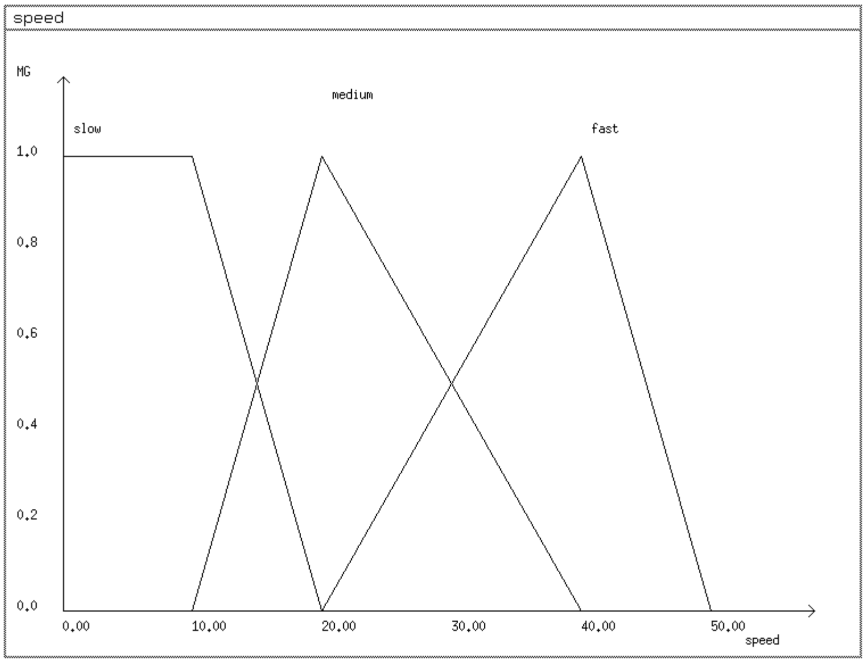

Figure 4: *Shown here are the membership functions of the fuzzy terms for speed displayed by using the window interface.*

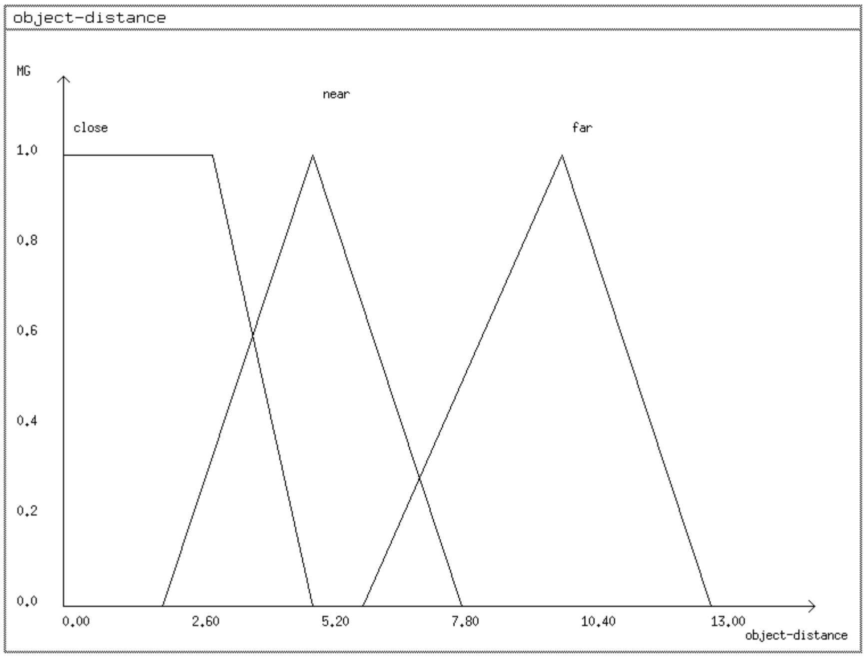

Figure 5: *Shown here are the membership functions of the fuzzy terms for object-distance displayed by using the window interface.*

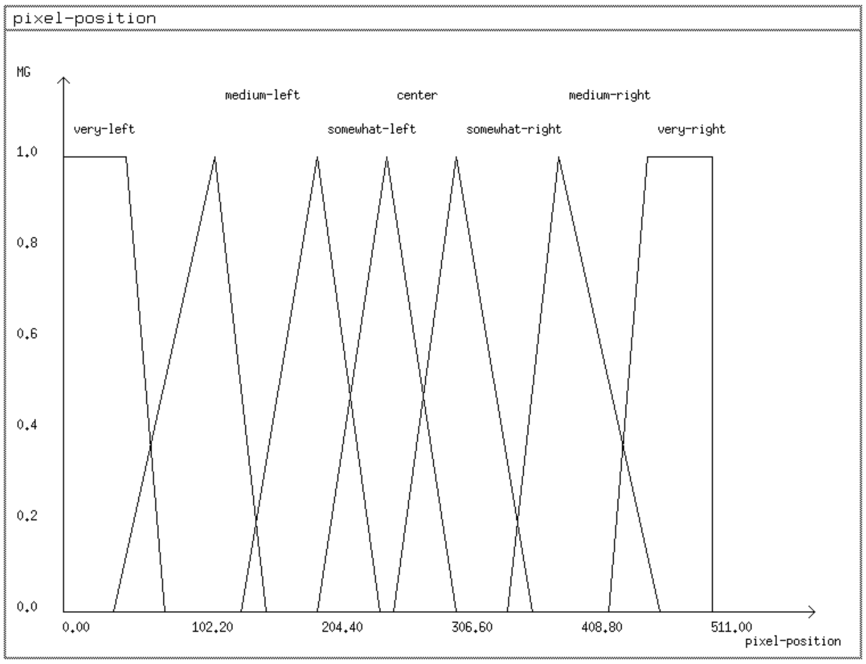

Figure 6: *Shown here are the membership functions of the fuzzy terms for pixel-position displayed by using the window interface.*

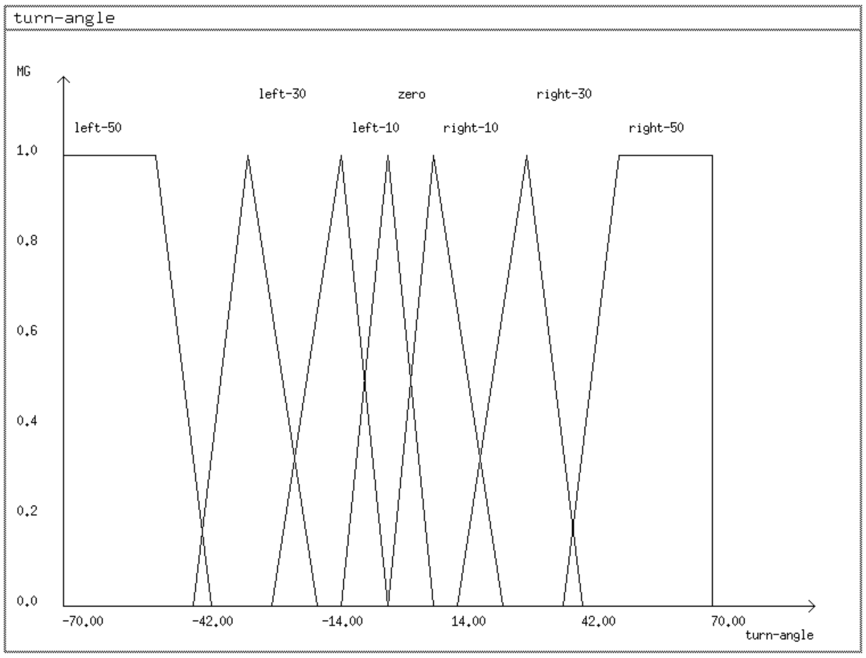

Figure 7: *Shown here are the membership functions of the fuzzy terms for turn-angle displayed by using the window interface.*

Another useful command is *degree-of-match*. This command shows the degree-of-match value between two given fuzzy terms.

Modifiers for Fuzzy Terms

The membership functions may be changed by the so-called modifiers such as *somewhat*, *more_or_less*, *very* and *extremely*. [5] The membership function of a fuzzy term with a modifier, say *very*, can be obtained by applying an arithmetic operation to the membership function of the fuzzy term. The arithmetic operations are listed below. Note that F(x) is the membership function of the term that needs to be modified.

Modifiers	Modifying Operation
somewhat	$F(x)^{\frac{1}{2}}$
more_or_less	$F(x)^{\frac{1}{3}}$
very	$F(x)^2$
extremely	$F(x)^3$

Shown in Fig.8 to Fig.11 are the membership functions obtained through these modifiers for the fuzzy term *far*. These membership functions were obtained by first sampling each linear segment of the original membership function at a number of points and then applying the operation to each such sampling point. Evidently, the precise form of the modified terms will depend on the number points at which each linear segment is sampled. A fuzzy term with a modifier can appear in the facts, also in the rule-antecedents. The user can assert a fact like (distance very far) or define a rule-antecedent like (distance somewhat near) or (robot (object-distance very far)(speed fast)).

Defining Fuzzy Facts

There exist two different types of facts in FuzzyShell. The first type is intended for facts that are about what we refer to as *disembodied* linguistic variables. Consider, for example, the fact (temperature high) for the linguistic variable *temperature*. This fact can be represented as such in the working memory. If at a later time, another fact such as (temperature low) is also deposited into the memory, the system will aggregate the evidence in the two facts and automatically create a single updated fact of the form (temperature aggregate_term), with the updated membership function assigned to the special term *aggregate_term*. Facts of this kind are declared by the syntax

(linguistic-variable fuzzy-term)   ;;; Type 1 fact

[5] *very-far* is a fuzzy term, but "very far" is the fuzzy term *far* with the modifier *very*.

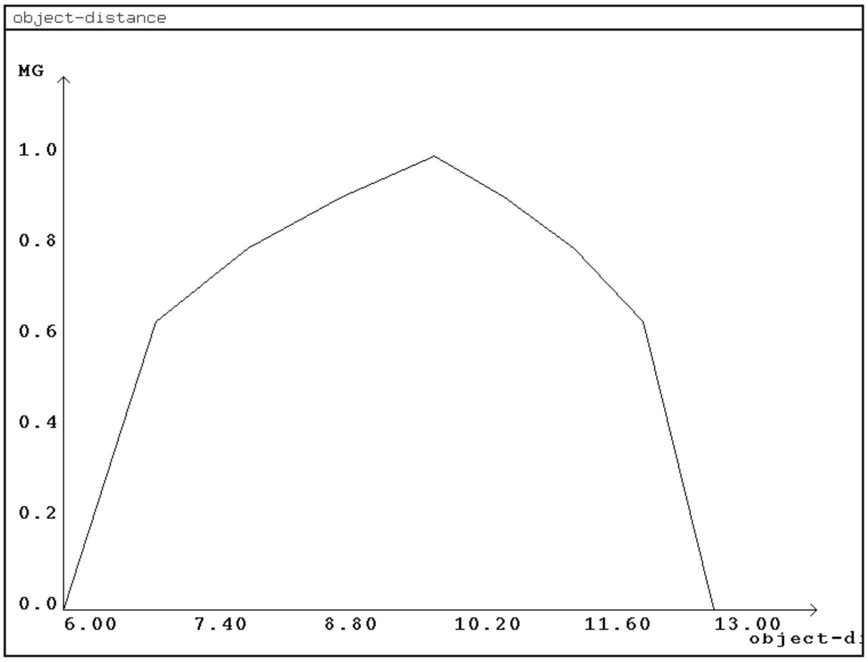

Figure 8: *Shown here the membership function for the fuzzy term far with the modifier somewhat.*

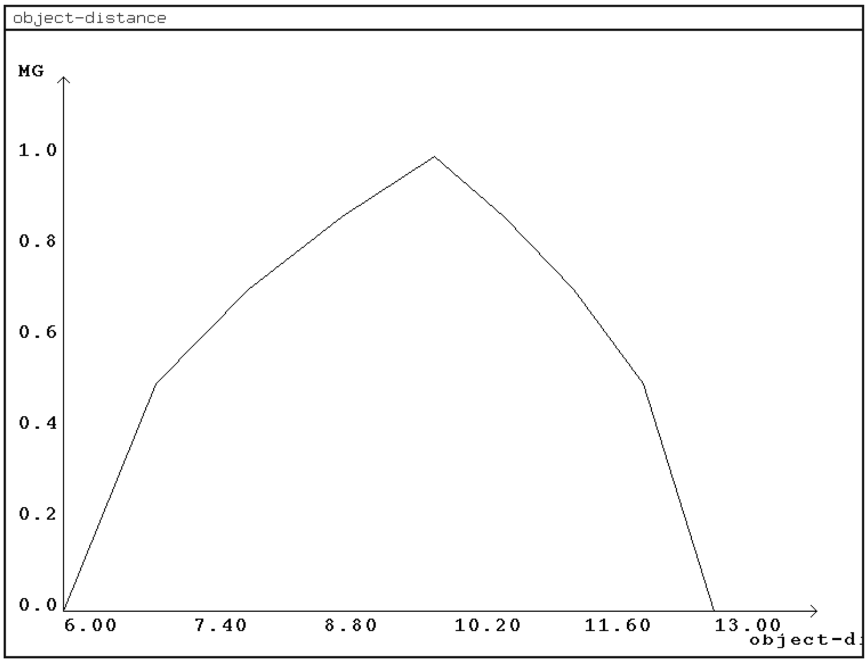

Figure 9: *Shown here the membership function for the fuzzy term far with the modifier more_or_less.*

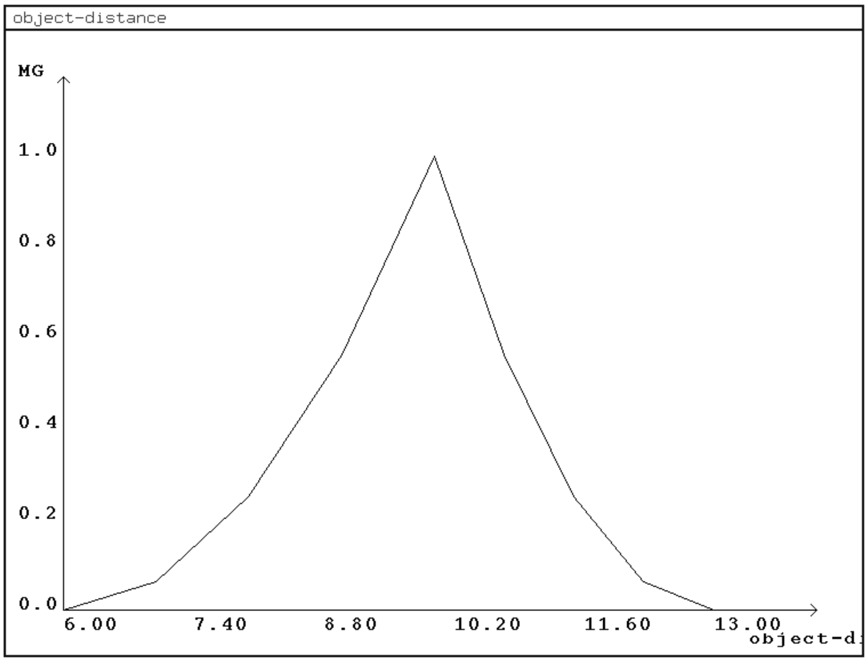

Figure 10: *Shown here the membership function for the fuzzy term far with the modifier very.*

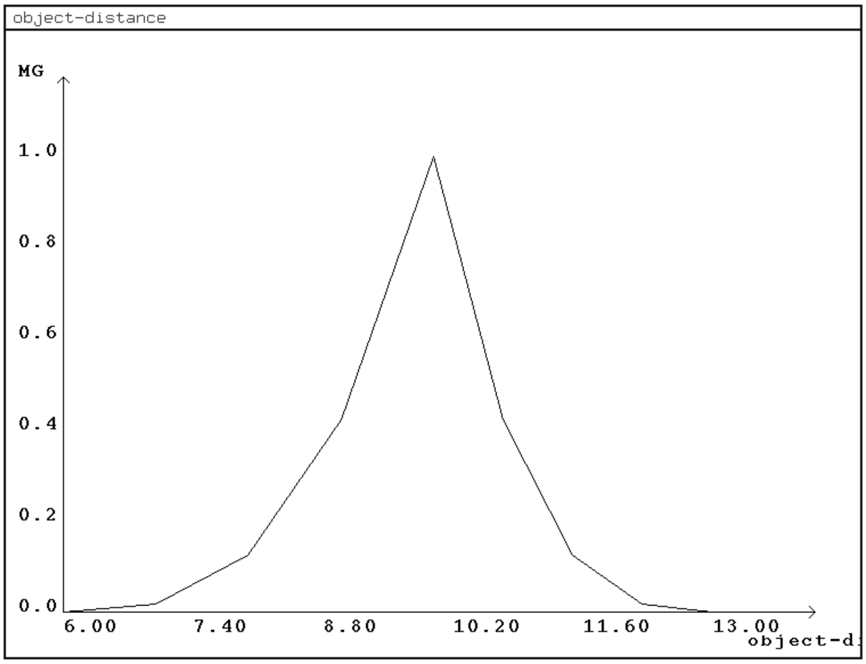

Figure 11: *Shown here the membership function for the fuzzy term far with the modifier extremely.*

Now consider the case of *embodied* facts, meaning facts that are about named objects. As an example, consider

```
(robot
    (name peter)
    (speed fast)
    (object-distance far))
```

This fact, containing information about the linguistic variables *speed* and *object-distance*, is with regard to a specific object. The object in question is of class *robot* and its name is *peter*. Evidently, when linguistic variables appear in embodied facts, they cannot be aggregated in the manner described above. The evidence (speed fast) in the above fact ought not to be combined with the evidence (speed low) in

```
(robot
    (name alvin)
    (speed low)
    (object-distance far))
```

From the syntax used when a fact is asserted into the working memory, FuzzyShell automatically knows whether or not aggregate the evidence for a given linguistic variable. For embodied facts, the evidence for a given linguistic variable is aggregated only if the the variable in question refers to the same object. As shown by the two examples above, a fact about a specific object is expressed using the syntax

```
(template-name      ;;; Type 2 fact
      (crisp-slot-1 slot-value-1)
                .
                .
      (crisp-slot-m slot-value-m)
      (linguistic-variable-1 fuzzy-term-1)
                .
                .
      (linguistic-variable-n fuzzy-term-n))
```

Facts of either type can be asserted into the working memory using any of the CLIPS utilities for this purpose. For example, at the top level, the *assert* command is usually used for this purpose:

```
command> (assert (robot (name peter)(speed fast)(object-distance far)
```

The same command can be used on the right hand side of a rule, as in this example:

```
(defrule rule5-a
    (distance near)
  ⇒
    (assert (speed fast)))
```

A group of facts can be asserted simultaneously into the working memory using the *deffacts* command at the top level:

```
(deffacts fact-name
    (distance near)
    (robot (name peter)(speed fast)(object-distance far)))
```

For asserting facts from external functions, the function *AssertSting* can be used, as in

```
AssertString("robot (name peter)(speed fast)(object-distance far)")
```

The operation above can be performed in a user's C program, and it is equivalent to asserting at the top level (assert (robot (name peter)(speed fast)(object-distance far))).

Before concluding this section, we will show how to display the membership function associated with the special term *aggregate_term*. Recall, whenever evidence is aggregated about any linguistic variable, the combined evidence results in a more general membership function than is defined for any of the terms associated with the linguistic variable. This new membership function is then assigned to the special term *aggregate_term*. In order to display this membership function, the user must specify the fact id associated with the updated linguistic variable.

For example, consider that the following four facts have been asserted into the working memory simultaneously:

Fact-a: (robot (n̂ame Peter)(ŝpeed fast))
Fact-b: (robot (n̂ame Alvin)(ŝpeed medium))
Fact-c: (robot (n̂ame Peter)(ŝpeed medium))
Fact-d: (robot (n̂ame Alvin)(ŝpeed slow))

FuzzyShell will automatically aggregate the fuzzy evidence in Fact-a and Fact-c on the one hand, and in Fact-b and Fact-d on the other. Therefore, the above four facts will give rise to the following two facts

Fact-1: (robot (n̂ame Peter)(ŝpeed aggregate_term))
Fact-2: (robot (n̂ame Alvin)(ŝpeed aggregate_term))

The fuzzy membership functions for the term *aggregate_term* for the robot Peter would be different from the membership function for the term *aggregate_term* for the robot Alvin. To display the former membership function, as shown in Fig.12, in the window of Deflingvar Manager the user needs to click on the linguistic variable *speed* and then click on the *SelectID* button to set the fact id to be 1. Finally, clicking on the *Draw* button shows the membership function for the *aggregate_term* of *speed*.

Another way to show the membership function of *aggregate_term* for any linguistic variable is through the use of the command *show-aggregate-term*. For example, to display the membership function for this term for the linguistic variable *speed* for the robot Peter, we'd say

command> (show-aggregate-term 1 speed)

FuzzyShell will show the following information in response to the command above.

(10.00 0.00) => (20.00 1.00) => (30.00 0.5)
=> (40.00 1.00) => (50.00 0.0)

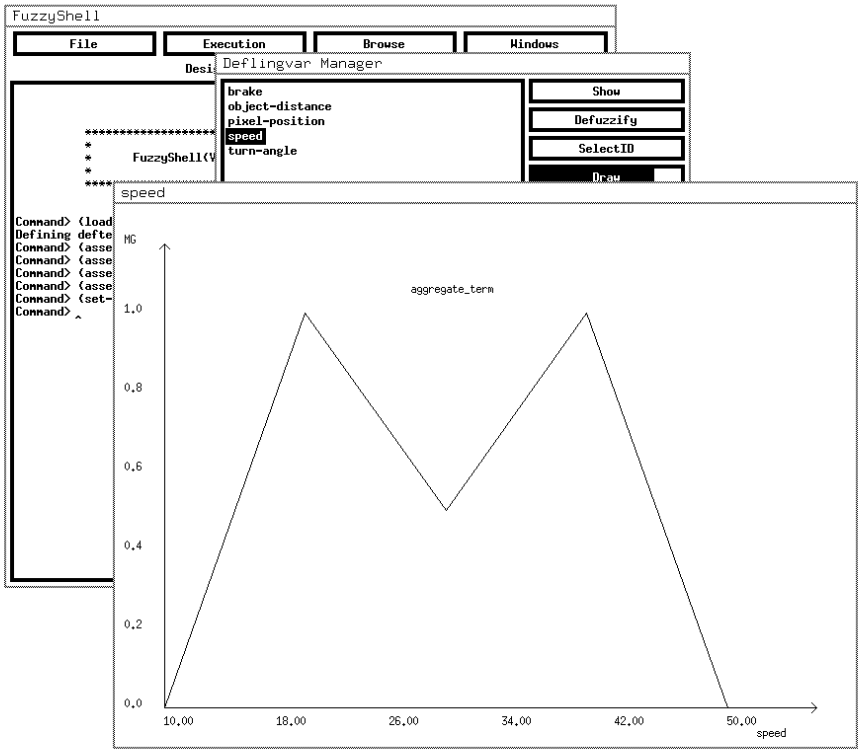

Figure 12: *Shown here is the membership function of aggregate_term for speed associated with the fact of id = 1 by using the window interface.*

5 Fuzzy Evidence Propagation through Rules

We will now explain how fuzzy pattern matching is carried out to determine what rules should be fired and how fuzzy evidence is propagated through the rules. For the purpose of matching facts with rule antecedents, FuzzyShell uses the well-known Sup-Min strategy. The use of this

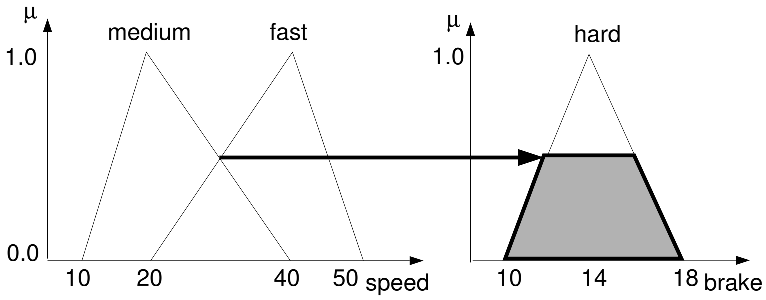

Figure 13: *The degree-of-match between the terms medium and fast is used to truncate the membership function of the term hard.*

strategy for pattern matching and the propagation of fuzzy evidence through rules will now be explained with a few examples. We will first consider the case of rules with single antecedent elements, followed by rules with multiple antecedents, and then by the rules involving variables.

First we consider the case of simple rules involving a single antecedent element:

A rule:

```
(defrule case1
    (speed fast)
  ⇒
    (assert (brake hard)))
```

A fact:

```
(speed medium)
```

Since the membership functions for the terms *medium* and *fast*, the fact shown will match with the antecedent of the rule. The degree-of-match associated with this rule calculated by using the Sup-Min formula, as shown in Fig. 13, will be used to truncate the membership function of the fuzzy term, *hard*, appearing in the consequent of the rule.

Before illustrating rules with multiple antecedents, we will now quickly show a small variation on the previous case, the case when the fact uses a single number as opposed to a fuzzy term. Consider

A rule:

```
(defrule case2
    (speed fast)
  ⇒
 (assert (brake hard)))
```

A fact:

```
(speed 26.5)
```

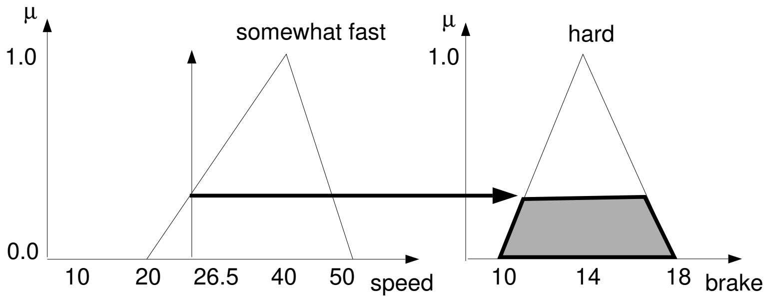

Figure 14: *The degree-of-match between fast and the number 26.5 is used to truncate the membership function of hard.*

The degree of match when the fact is matched with the rule antecedent is shown in Fig.14. As before, this degree of match is used to truncate the membership function of *hard*.

Now let's consider the case of a rule with two antecedent elements. Our discussion easily generalizes to rules with an arbitrary number of antecedent elements.

A rule:

```
(defrule rule9-a
    (robot (object-distance far)(pixel-position medium-left))
    (speed medium)
  ⇒
    (assert (brake hard)))
```

Facts:

```
(robot (object-distance near)(pixel-position very-left))
(speed fast)
```

The first antecedent element contains two fuzzy term, *far* and *medium-left*, and the second antecedent element contains one fuzzy term, *fast*. The tenets of fuzzy logic dictate that the minimum of the degrees of match between these three fuzzy terms in the rule-antecedents and the corresponding fuzzy terms in the facts be used to truncate the membership function of the consequent fuzzy term(s), in this case the term *hard*. This is illustrated in Fig.15.

We next discuss the case of multiple rules, all containing the same linguistic variable in their consequents. In the example to follow, given the fact (distance close), the first two rules must both be fired and fuzzy evidence about the consequent fuzzy term aggregated before the third rule is allowed to fire. As was discussed earlier, FuzzyShell ensures this kind of operation through through a new kind of Rete network. FuzzyShell aggregates the truncated versions of the membership functions for the consequent terms *slow* and *fast*, the truncation levels being determined by the degrees of match between the fact term *close* and the antecedent terms, as shown in Fig.16. The aggregated membership function is assigned to a special term called *aggregate_term*

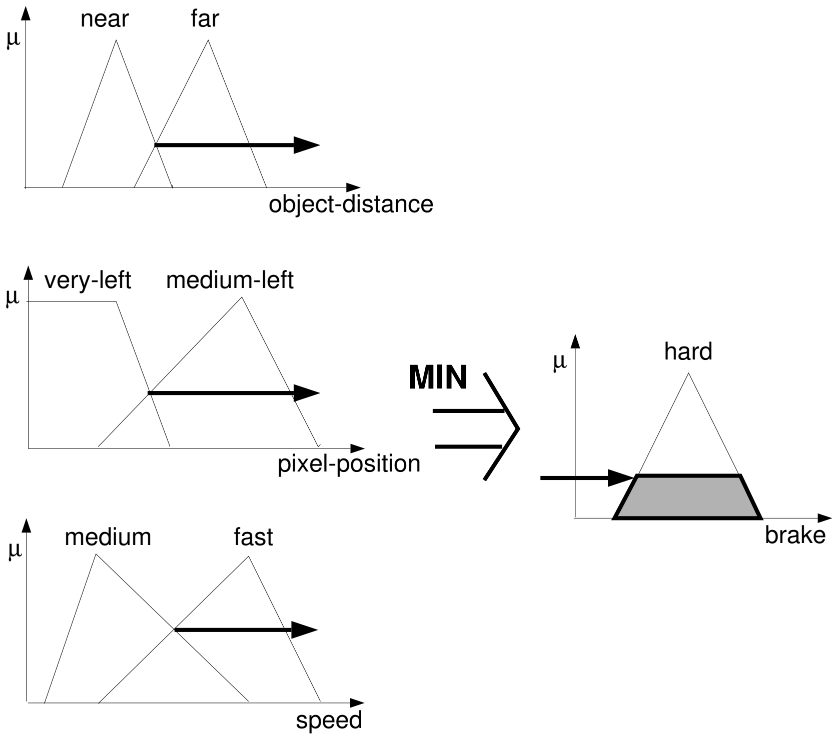

Figure 15: *The minimum of degree-of-matches for the fuzzy terms in the rule-antecedents is used to truncate the membership function of hard.*

for the linguistic variable *speed*. [6] Subsequently, since there is an overlap with the computed membership function for *aggregate_term* and that for the term *fast*, the inference engine discovers that rule11-c can be fired. The firing of this rule results in the truncation of the membership function for the term *hard* for the linguistic variable *brake*, as shown in Fig.17. Note that this truncated membership function will be assigned to the term *aggregate_term* for the linguistic variable *brake*.

Rules:

```
(defrule rule11-a
    (distance far)
  ⇒
    (assert (speed fast)))

(defrule rule11-b
    (distance close)
  ⇒
 (assert (speed medium)))

(defrule rule11-c
    (speed fast)
  ⇒
 (assert (brake hard)))
```

A fact:

```
(distance near)
```

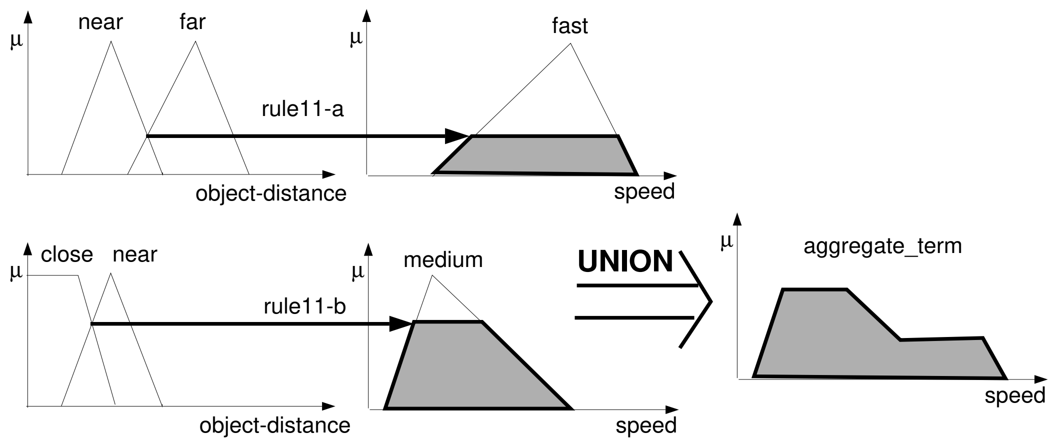

Figure 16: *The evidence of speed inferred by rule11-a and rule11-b is aggregated by the union operator.*

[6]This issue is discussed in great detail in the research publication that accompanies this manual.

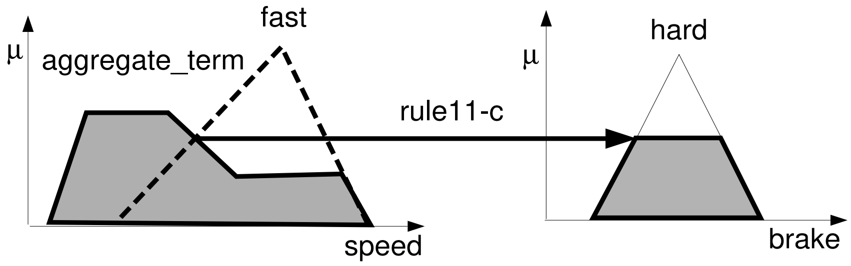

Figure 17: *The degree-of-match between fast and the term aggregate_term of speed will be used to truncate the membership function of hard.*

We will now discuss an example that exercises the Join Network part of the Rete Network. (Details regarding the Join Network are in the accompanying research paper. Basically, the job of a Join Network is to ensure that the bindings assigned to the variables in the different antecedent elements of a rule are consistent.) Consider a rule that has the same variable ?x appearing in the two different condition elements:

```
(defrule rule12
    (Robot (name Peter)(object-distance ?x))
    (Robot (name Alvin)(object-distance ?x))
  ⇒
    ( . . .))
```

```
facts:
        (Robot (name Peter)(object-distance far))
        (Robot (name Alvin)(object-distance near))
```

As is evident from the two facts, the variable ?x in the first condition element will be bound to the fuzzy term *far*, while the same variable in the second condition element will be bound to the term *near*. Now, in a traditional expert system shell, this state of affairs will cause the rule shown to be not firable given the two facts above. However, the Join Network in FuzzyShell correctly decides that the rule can be enabled despite the disparate bindings for the same variable on account of the overlap between the membership functions for *far* and *near*. As far as the truncation of the membership function of any consequent fuzzy terms is concerned, in keeping with the principles of fuzzy logic the truncation level set equal to the degree-of-match between *far* and *near* will be used to truncate the membership functions of fuzzy terms.

Of course, the above discussion generalizes to any number of condition elements in a rule with any number of variables in each of the condition elements.

6 Defuzzification

Defuzzification in FuzzyShell is based on the "center of gravity" method. The formula for defuzzification is expressed as follows.

$$Defuzzified\,Value = \frac{\int_{x \in U} (x \cdot F(x)dx}{\int_{x \in U} F(x)dx}$$

$$approximate\ defuzzified\ value = \frac{\sum_{i=1}^{n} x_i \cdot F(x)}{\sum_{i=1}^{n} F(x)}$$

FuzzyShell uses sampled points to obtain the approximated defuzzified value. The number of sampling points used for defuzzification is specified by the user. We typically use 100 points.

The syntax of the defuzzify command is

(defuzzify fact-id linguistic-variable)

The first parameter, *fact-id*, is the fact id or fact address where the linguistic variable resides. The second parameter, *linguistic-variable*, is to specify which linguistic variable needs to be defuzzified. For example, consider the following three facts.

Facts:
fact-3:(robot (name Peter) (distance far)(velocity fast))
fact-4:(robot (name Alvin)(distance near)(velocity slow))
fact-5:(velocity fast)

If the user wants to defuzzify the linguistic variable *velocity* for fact-4, then the first parameter must be 4 and the second must be *velocity*. We will use the following example to illustrate how to use the *defuzzify* command in a rule.

Example 13

```
(defrule rule13-a
    (robot (name peter)(distance far))
  ⇒
    (assert (speed fast)))

(defrule rule13-b
    (robot (name peter)(distance close))
  ⇒
    (assert (speed slow)))
```

```
(defrule rule13-c
      ?f ← (speed ?x)
    ⇒
      (bind ?result (defuzzify ?f speed)))
```

Assume there is a fact (robot (name peter)(distance near)) to activate rule13-a and rule13-b. In the first cycle of the inference engine, the evidence posited by rule13-a and rule13-b for *speed* will be aggregated. Subsequently, the aggregation result will be used to trigger rule13-c. The purpose of rule13-c is to defuzzify *speed* and then set the variable *?result* to a numerical value returned by the defuzzification function. The user can also issue the *defuzzify* command through the window interface. For this purpose, the user must first pop up the window for *Deflingvar Manager* under the *Browse* menu. Then, the user types in the fact-id by using the *SelectID* button. Subsequently, the linguistic variable desired is then clicked, followed by a click on the *Defuzzify* button.