Conflict Resolution in Fuzzy Forward Chaining Production Systems

James Bowen
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA.

Jianchu Kang
Department of Computer Science
Institute of Aeronautics and Astronautics
Beijing, Peoples Republic of China

Abstract

Forward-chaining productions have been used to implement some of the most significant expert systems. However, most forward chaining production languages make no provision for dealing with lexical imprecision. This paper briefly presents a language which supports fuzzy matching between condition patterns and facts in working memory. Then discussion is focussed on what part should be played in conflict resolution by the relative truth-values of fuzzily matched production instantiations.

1 Introduction

Forward-chaining productions have been used to implement some of the most significant expert systems [McDermott, 1980]. However, most forward-chaining production languages make no provision for handling lexical imprecision. Thus, for example, there is no notion of fuzziness in such languages as OPS5 [Brownston et al., 1985], OPS83 [Forgy, 1983], ART [Inference Corporation, 1986], YAPS [Allen, 1983], YES/L1 [Milliken et al., 1985], or YES/OPS [Schor et al., 1986].

The fuzzy reasoning literature contains many references to fuzzy production systems [Togai and Watanabe, 1986]. Since these systems are data-driven, they may be viewed as forward-chaining. However, they are different from the classical type of forward-chaining production system.

On each cycle in the classical type of forward-chaining system, only a subset of the productions have satisfied condition parts, and of these, only a subset, typically one, is fired. A conflict resolution algorithm is used to determine which of the satisfied productions should be fired. On the other hand, in a fuzzy production system [Whalen and Schott, 1983], all productions can be considered as firing during each cycle (but with strengths ranging along a continuum from "not at all" to "completely"), rather than a subset of productions firing in the all or nothing fashion characteristic of classical forward-chaining production systems. The theme of this paper is bridging the gap between classical forward-chaining production languages and fuzzy production systems, by extending classical production languages to handle lexical imprecision.

The main purpose of this paper is to discuss the relationship between fuzzy reasoning and a key aspect of forward-chaining productions, namely conflict resolution. All examples are couched in an experimental forward-chaining production language, called FMUFL, which supports the use of lexical imprecision. In Section 2, the handling of lexical imprecision in FMUFL is briefly described. In Section 3, some issues involved in conflict resolution in general are presented; then attention is focussed on issues particular to doing conflict resolution in fuzzy production languages. The strategy used in FMUFL is described and contrasted with the approach used in FLOPS [Buckley et al., 1986; Siler et al., 1987], the only other fuzzy forward-chaining production system language known. Some conclusions are presented in Section 4.

2 Lexical Imprecision in FMUFL

FMUFL was derived by adding provision for lexical imprecision to an earlier language called MUFL [Bowen, 1986; Bowen, 1987; Bowen, 1988]. Fuzzy reasoning in FMUFL is based on both finite and infinite fuzzy sets. Finite sets are described by exhaustively listing those elements of the universe of discourse which have non-zero degrees of membership. Infinite fuzzy sets are represented as I-type sets [Baldwin and Zhou, 1984].

I-type sets are used to represent concepts involving an underlying variable which is amenable to interpolation. For example, the following declaration defines the fuzzy concept *expensive*:

table(expensive,food,[0,200],
[[20,0],[50,0.5],[70,0.8],[100,1]]) (1) [1]

Here, the universe of discourse, *[0,200]*, is a segment of the real number line representing prices of food, and *expensive* is the label for a fuzzy subset of this universe. The fuzzy set is defined by giving an ordered list of points from the universe of discourse and associating with each point its membership grade in the set, in the interval [0,1]. If the membership degree in the fuzzy set of two prices is known, the membership degree of some intervening price can be interpolated.

Finite sets are used to represent concepts such as *hungry* which do not involve an underlying variable amenable to interpolation. These sets are declared by exhaustively listing their members, giving the degree to which each is a member of the set. For example, the following defines the fuzzy set *hungry* which is a fuzzy subset of some discourse universe of people:

hungry(mary) **with truth** 0.8 (2)
hungry(tom) **with truth** 0.5 (3)
hungry(john) (4)

[1] Bold face indicates an FMUFL reserved word.

FMUFL also provides a mechanism, called a semantic relationship, which can be used to ease the burden of defining fuzzy sets. If a fuzzy set, be it an I-type set or a finite set, has been explicitly defined to represent some concept, other fuzzy sets which represent related concepts can be defined by declaring semantic relationships which specify how the concepts are related. FMUFL supports several types of semantic relationships, including antonyms and synonyms. Suppose, for example, that the concept *expensive* is defined as in (1) above. If *cheap* is defined as an antonym of *expensive*, as follows

antonym(expensive, cheap) (5)

FMUFL will treat the concept *cheap* as equivalent to *not expensive*. That is, it will treat *cheap* as if it were defined by the following table:

table(cheap,food,[0,200],
[[20,1],[50,0.5],[70,0.2],[100,0]]) (6) [2]

Similarly, defining

modified_synonym([ravenous, **very** hungry]) (7)

means that FMUFL will treat *ravenous* as synonymous with the hedged linguistic value *very hungry*; that is, as if it were defined by the following:

ravenous(mary) **with truth** 0.64 (8)
ravenous(tom) **with truth** 0.25 (9) [3]
ravenous(john) (10)

Fuzzy propositions involving concepts, such as *hungry*, which must be specified through exemplification rather than interpolated, are represented in working memory as members of finite fuzzy sets. Thus, for example, the working memory entry (2) which might represent the descriptive proposition

Mary is quite hungry

is treated as specifying the membership degree of *mary* in the finite fuzzy set *hungry*. Similarly, the entry

likes(mary, bread) **with truth** 0.7 (11)

which could be used to represent the relational proposition

Mary rather likes bread

is treated as specifying the membership degree of the pair *(mary, bread)* in the finite fuzzy set *likes*. These working memory facts can be used, with semantic relationship definitions, to match production condition patterns such as (12) and (13)

ravenous(Who) (12) [4]
likes(mary,What) (13)

Condition patterns involving a variable, such as price, whose values are amenable to interpolation, are handled using I-type fuzzy sets. Thus the working memory fact

price(bread, food, 35) (14)

can be used, in conjunction with the I-type sets defined by (1) and (5), to match, with different truth values, production condition patterns such as (15) and (16).

price(bread, food, expensive) (15)
price(What, food, **very** cheap) (16)

Just as the LHS of a production can contain fuzzy queries and/or match fuzzy propositions in working memory, so its RHS can assert/retract fuzzy propositions to/from the working memory. Consider production (17).

when ravenous(P) **and** likes(P,F) **and**
price(F,food, **fairly** cheap) (17)
then **store** should_buy(P,F) **qualified**

This states that whenever it is possible to find a pair of entities which satisfy the composite fuzzy query

ravenous(P) **and** likes(P,F)
and price(F,food, **fairly** cheap) (18)

then it should be asserted in the working memory that the pair of entities belong to the finite fuzzy relation *should_buy*, the membership degree being equal to the truth value of the composite fuzzy query in the LHS of the rule. Based on working memory facts (2), (11) and (14), the following instantiation of production (17) would enter the conflict set:

when ravenous(mary) **and** likes(mary,bread) (19)
and price(bread,food,**fairly** cheap)
then **store**
should_buy(mary,bread) **qualified**

The membership degree of (19) in the conflict set is the same as the truth value of the LHS, that is 0.64, which is calculated as follows.

The truth value of *ravenous(mary)* is 0.64, based on the following: the truth value of *hungry(mary)* is 0.8, from (2); *ravenous* is *very hungry* from (7); *very 0.8* is 0.64. The truth value of *likes(mary, bread)* is 0.7, from (11). The truth value of *price(bread, food,* **fairly** *cheap)* is 0.87, based on the following: the price of bread is 35 from (14); the membership of 35 in *expensive* is 0.25, based on interpolation between the membership grades given in (1) for 20 and 50; *cheap* is *not expensive*; *fairly cheap* is *fairly not expensive*; *fairly not 0.25* is $(1 - 0.25)^{0.5} = 0.87$. [5] The overall truth value of the LHS of the instantiation is derived from these constituent truth values, by interpreting logical *and* as min; that is $0.64 \wedge 0.7 \wedge 0.87 = 0.64$.

[2] The effect of the **not** modifier is to complement membership grades.

[3] The effect of the **very** hedge is to square membership grades.

[4] Tokens, like *Who* and *What*, which start with upper case letters are variables.

[5] The **fairly** hedge returns the square root of a membership grade.

3 Conflict Resolution in Forward chaining Production Systems

Usually, more than one production is satisfied on any one cycle of a forward-chaining production system and frequently some of these productions may have several instantiations. A conflict-resolution strategy is a coordinated set of principles for selecting, among competing production instantiations, a subset to be executed. In most systems, only one production instantiation is executed on each cycle, although there are some systems [Siler et al., 1987] which may execute several instantiations per cycle.

Conflict resolution is of vital importance in a forward-chaining production system because it influences two crucial aspects of the system [Brownston et al., 1985; McDermott and Forgy, 1978] : its *sensitivity* and its *stability*. A system that is responsive to the demands of its environment is said to display sensitivity. One that is able to maintain continuity in its behaviour is said to display stability. Of these two characteristics, sensitivity is the more important; it is what distinguishes the forward-chaining production paradigm from other computational models. There are several kinds of sensitivity. A system should be sensitive not just to the contents of, but also to changes in, its working memory. Even more importantly, a forward-chaining interpreter should also be sensitive to its own state; if there is some state information available which indicates that the system is about to enter an infinite loop, the interpreter should immediately take account of this information, to avoid looping.

A conflict resolution strategy may be viewed as a series of seives. Production instantiations are "poured" into the topmost seive, those that filter through being passed on to the next seive, and so on, until an acceptable set of firable instantiations (typically of cardinality 1) is produced. The interpreters for different production system languages use different seives. The choice of seives to be used in a conflict resolution strategy, and the order in which they are to be applied, depends on the class of problem for which the production system language is intended. Some languages [Forgy, 1983] allow the programmer to design his own conflict resolution strategy.

Though OPS5 is now a relatively old forward-chaining language, as a default strategy for general purpose programming, its conflict resolution strategy (or strategies, since two variants are provided) is still the most valid. The MEA variant of this strategy is particularly useful, since it supports task-oriented programming [Brownston et al., 1985]. Consequently, when designing the conflict resolution strategy for FMUFL the OPS5 strategy was chosen as a basis. The FMUFL strategy was to be upwardly compatible with the OPS5 strategy: when no lexical imprecision was present, the FMUFL strategy was to be the same as that for OPS5.

The OPS5 strategy consists of the following five seives, applied in the order given: refraction; relative recency; relative element specificity; relative test specificity; arbitrary choice. (However, recency and element specificity are not really separated; they are implemented by the same code.) Refraction means that an instantiation should be removed from the conflict set if it has fired on a previous cycle and if it has been present in the conflict set on each cycle since it last fired. Relative recency specifies that, of the instantiations remaining in the conflict set, all should be removed except those which match the most recently asserted of all those facts which are matched by any instantiation in the conflict set. Relative element specificity means that, when comparing two instantiations, preference should be given to the one which is based on a larger subset of the facts (elements) in working memory. Relative test specificity dictates that, of the remaining instantiations, preference should be given to those with the greatest number of tests in the LHS. Arbitrary choice is only used if the previous seives have failed to reduce the conflict set down to one instantiation: an instantiation is chosen at random from among those remaining.

The conflict resolution strategy in a fuzzy language must also consider the absolute and relative truth-values of instantiations. An absolute truth-value seive would prevent, from entering the conflict set, any instantiations which have a truth-value below some threshold. A relative truth-value seive would allow only those instantiations which have the highest membership grade, of all those remaining in the conflict set, to pass through to the next seive. So, in designing the conflict resolution strategy for FMUFL, it was necessary to determine where to place these truth-value seives in the sequence. In FMUFL, the default threshold applied to absolute truth-values is 0.5, but this can be altered by the programmer; the appropriate position for this seive is obvious: it should be applied first, even before refraction, since its function is to prevent instantiations from entering the conflict set at all.

However, the correct position for the relative truth-value seive is less obvious. There seems to be only one other fuzzy forward-chaining production system language, namely FLOPS [Buckley et al., 1986, Siler et al., 1987]. There are two versions of FLOPS, a serial version [Buckley et al., 1986] in which only one instantiation fires per cycle, and a parallel version [Siler et al., 1987] in which several instantiations may fire per cycle. In the serial version of FLOPS, which is also based on OPS5, the relative truth-value seive is the first seive applied in conflict resolution. However, based on our perception of the need to support the task-oriented programming methodology commonly advocated [Brownston et al., 1985] for forward-chaining productions, the relative truth-value seive should be applied much later in conflict resolution.

There were six possible positions for this seive, marked (a) through (f) below.

(a) ⇒
    refractoriness
(b) ⇒
    recency
(c) ⇒
    element specificity
(d) ⇒
    test specificity
(e) ⇒
    arbitrary choice.
(f) ⇒

Position (c) does not really exist in languages with an OPS5-like conflict resolution strategy where both recency and element specificity are implemented by the same code. However, the position is identified here, to make the point

that recency supports a second order sensitivity, (it is sensitive to *changes* in the state of working memory), whereas element specificity provides only first order sensitivity (to the *contents* of working memory). It turns out, however, that position (c) is not appropriate for the relative truth-value seive anyhow, as will be seen below.

Position (a) was rejected since the refractoriness criterion ought to be first because it protects the system from infinite loops. Arbitrary choice should be the principle of last resort, so position (f) would be pointless.

Position (b) was rejected, based on the following reasoning. The truth-value of an instantiation reflects the compatibility between the state description in working memory and the (possibly abstract) state description in the LHS of the production on which the instantiation is based. In this respect, truth-value is similar to test specificity and contributes to the stability of the system rather than to its sensitivity, although like element specificity it could also be regarded as contributing to first order sensitivity. Since the recency seive contributes to the second order sensitivity of the system, recency ought to precede truth-value in conflict resolution.

Position (c) had to be eliminated in order to meet a primary aim in the design of FMUFL: to ensure that methodologies which have evolved for programming in crisp forward-chaining languages should also be usable in FMUFL. The most important such methodology is the idea of task oriented programming [Brownston et al., 1985; Bowen, 1987], in which each production is associated with a particular task in a hierarchy and has, as its first condition pattern, a test for the presence in working memory of a flag which indicates that the task is active. Task-activation flags are asserted into and removed from working memory in much the same way as activation records are pushed onto and popped from a stack during the execution of a program written in a traditional block-structured procedural language.

In order to support the task-oriented programming methodology, element specificity must be applied before truth-value in conflict resolution. This can be seen by considering a fragment from a program that implements an extended version of the grocery configuration problem [Winston, 1984] which is commonly used to explain task-oriented programming. The grocery problem is extended to include lexical imprecision by specifying that items, which complement groceries already selected, should only be added to the selection if they are cheap, where *cheap* is fuzzily defined, as in (1) and (5). The task of adding complementary items is implemented as a collection of productions like (20),

```
when  doing(add_cheap_extras) and
      selected(List) and potato_chips in List
      and untrue(pepsi in List)
      and price(pepsi,food,cheap)
then  make NewList = pepsi plus List and      (20)
      replace selected(List)
      by selected(NewList)
```

each of which checks for a situation in which an item should be added. Additionally, a production like (21) is needed,

```
when  doing(add_cheap_extras)
then  remove doing(add_cheap_extras)    (21)
```

to terminate the task by removing from working memory the flag which indicates that the task is active; this production should fire only after all satisfied productions like (20) have been executed.

Consider the point where some other production has just stored in working memory a flag to activate this task. The relevant working memory elements, with their associated time-tags might look like this:

price(pepsi, food, 35)	time tag 4	(22)
selected([bread,jam,potato_chips])	time tag 30	(23)
doing(add_cheap_extras)	time tag 31	(24)

The instantiation of (20) would only have a truth-value of 0.75, based on the membership of 35 in the fuzzy set *cheap*, while the instantiation of (21) would have a truth-value of 1; basing a choice between these two instantiations on relative truth-values would, therefore, prevent the addition of pepsi to the grocery selection. Indeed, the only items that could ever be added to the selection are those with prices having a membership grade of 1 in the fuzzy set *cheap*. By contrast, if relative element specificity were applied before relative truth-value, the instantiation of (20) would dominate, giving the desired behaviour. Relative element specificity must, therefore, be used before relative truth-value. Otherwise, the facility for handling lexical imprecision is eliminated; tasks will be terminated before fuzzily satisfied productions for performing the tasks have a chance to act. Position (c), therefore, is not appropriate for the truth-value seive.

There remains the choice between positions (d) and (e). Arguments can be advanced in favour of both positions. An argument in favour of (d) could be as follows. The truth-value of an instantiation depends on the working memory items matched, so using it contributes to the first-order sensitivity of a forward-chaining system. The test specificity of an instantiation depends on the underlying production, not on the data in working memory, so it does not contribute to sensitivity, but to stability. Sensitivity is more important than stability, so truth-value should be considered before test specificity. But a sensitivity-based argument could also be made against (d). For the sake of brevity, however, this will not be presented here.

Instead, noting that the choice between positions (d) and (e) is not clear cut, we chose position (e) for the following pragmatic reason. The conflict resolution strategy provided by a language is a tool to be used by programmers. Apart from arbitrary choice, which is a conflict resolution principle of last resort, the strategy should produce program behaviour which is *easily predictable* by both the author and the reader of a program. Furthermore, a conflict resolution strategy should enable the programmer to achieve a particular flow of control if he has a specific one in mind. The programmer can utilize the test specificity seive to fine-tune his program by adding extra tests to a particular production so as to enable one of its instantiations to fire ahead of those of some other production. The truth-value seive, however, cannot be exploited in the same way. It is usually very difficult to predict the overall truth value of a run-time instantiation while a program is being written, especially when truth values may depend on

user input. Thus, based on the need to support programmer determination of execution flow, test specificity ought to precede truth-value in conflict resolution; that is, the truth-value seive should be placed in location (e) above.

The differing approaches taken to conflict resolution in FMUFL and FLOPS means that these two languages are suitable for different classes of application. The simultaneous firing of several instantiations in the parallel version of FLOPS gives this version of the language some of the flavour of the production systems described in the fuzzy reasoning literature [Whalen and Schott, 1983]. This version of the language may be appropriate for fuzzy process control applications but parallel firing of instantiations prevents the type of execution flow required for task-oriented programming.

The rationale underlying the choice of conflict resolution strategy for serial FLOPS is not clear from publications on the language. However, since the parallel version of FLOPS (which is the newer version) is presented as a more efficient version of the language, this would indicate that serial FLOPS is also intended for problems which have much in common with fuzzy control applications. However, it is clear that the conflict resolution strategy selected means that this version of the language also cannot be used to write programs based on the task-oriented methodology. In FMUFL, however, the conflict resolution strategy was designed expressly to ensure that a task-oriented programming methodology could be supported.

4 Conclusions

Forward-chaining production languages are very powerful programming tools, as evidenced by their widespread usage. The expressive power of this class of language can be enhanced by enabling them to handle lexical imprecision. A method for doing this, based on fuzzy sets, was presented in this paper. This was followed by an analysis, based on the need to support task-oriented programming, of how to handle instantiation truth-values during conflict resolution. A conflict resolution strategy for fuzzy forward-chaining production system languages was developed and contrasted with conflict resolution in the only other fuzzy forward-chaining production language known.

References

[Allen, 1983] E Allen. YAPS: a production system meets objects. In *Proceedings AAAI-83*, pages 1-7, American Association for Artificial Intelligence, August 1983.

[Baldwin and Zhou, 1984] J F Baldwin and S Q Zhou. A fuzzy relational inference language, *Fuzzy Sets and Systems*, 14, pages 155-174, 1984.

[Bowen, 1986] J A Bowen. MUFL: A Multi-Formalism Language for Knowledge Engineering Applications. Technical Report, Department of Computer Science, North Carolina State University, 1986.

[Bowen, 1987] J A Bowen. Knowledge representation for partly-structured problems. In *Methodologies for Intelligent Systems*. Z W Ras (ed.). Elsevier, New York, 1987.

[Bowen, 1988] J A Bowen. A multiple paradigm language for supporting clarity of expression in expert systems. In *Proceedings 3rd. International Conference on Applications of Artificial Intelligence in Engineering*, August 1988.

[Brownston et al., 1985] L Brownston, R Farrell, E Kant and N Martin. *Programming Expert Systems in OPS5*. Addison-Wesley, Reading, MA, 1985.

[Buckley et al., 1986] J J Buckley, W Siler, and D Tucker. A fuzzy expert system. *Fuzzy Sets and Systems*, 20, pages 1-16, 1986.

[Forgy, 1983] C L Forgy. OPS83 Report. Technical Report, Department of Computer Science, Carnegie-Mellon University, 1983.

[Inference Corporation, 1986] Inference Corporation. *ART Reference Manual*. Inference Corporation, Los Angeles, CA, 1986.

[McDermott and Forgy, 1978] J McDermott, and C Forgy. Production system conflict resolution strategies. In *Pattern-Directed Inference Systems*. D A Waterman and F Hayes-Roth (eds.). Academic Press, New York, 1978.

[McDermott, 1980] J McDermott. R1: A Rule-Based Configurer Of Computer Systems. Technical Report, Department of Computer Science, Carnegie-Mellon University, 1980.

[Milliken et al., 1985] K R Milliken, A V Cruise, R L Ennis, J L Hellerstein, M J Masullo, M Rosenbloom and H M VanWoerkom. YES/L1: A Language for Implementing Real-Time Expert Systems. Research Report RC 11500 (#51654), IBM Thomas J Watson Research Center, Yorktown Heights, New York, 1985.

[Schor et al., 1986] M I Schor, T P Daly, H S Lee and B R Tibbitts. Advances in RETE Pattern Matching. In *Proceedings AAAI-86*, Science Section, pages 226-232, American Association for Artificial Intelligence, August 1986.

[Siler et al., 1987] W Siler, D Tucker and J Buckley. A parallel rule firing fuzzy production system with resolution of memory conflicts by weak fuzzy monotonicity, applied to the classification of multiple objects characterized by multiple uncertain features. *International Journal of Man-Machine Studies*, 26, pages 321-332, 1987.

[Togai and Watanabe, 1986] M Togai and H Watanabe. Expert system on a chip: an engine for real-time approximate reasoning. *IEEE Expert*, pages 55-62, Fall 1986.

[Whalen and Schott, 1983] T Whalen, and B Schott. Issues in fuzzy production systems. *International Journal of Man-Machine Studies*, 19, pages 57-71, 1983.

[Winston, 1984] P W Winston. *Artificial Intelligence*. Addison-Wesley, Reading, MA, 1984.