

# Implementing Fril++ for Uncertain Object-Oriented Logic Programming

J.F. Baldwin, T.H. Cao<sup>1</sup>, T.P. Martin and J.M. Rossiter

AI Group

Department of Engineering Mathematics

University of Bristol

United Kingdom BS8 1TR

{Jim.Baldwin, Tru.Cao, Trevor.Martin, Jonathan.Rossiter}@bristol.ac.uk

## Abstract

Uncertain object-oriented logic programming is a combination of logic programming, object-oriented programming and uncertainty logic to exploit the advantages of all three disciplines in dealing with real world problems. This paper presents the implementation of Fril++, the Fril-based object-oriented logic programming language with uncertainty. Fril++ syntax is introduced and related semantic issues, particularly, ones of uncertain, multiple, and overriding inheritance, are discussed and solutions to them are proposed. The implemented translator converting a Fril++ source program to a Fril target program is then presented with examples.

## 1 Introduction

It is now widely accepted that taxonomic information, and in particular class hierarchies, is an important part of a knowledge base. This is not only because objects in the real world are naturally associated with classes, but also because taxonomic information helps to organise knowledge through classification and provide efficient computation through inheritance. For development of such a system, object-oriented programming has provided advantageous methodologies and techniques. This is why extension of the classical object-oriented model with uncertainty, which is another essential aspect of knowledge, has been the quest and focus of significant research effort, e.g. [9], [10], [5], [13], [15], [4] and [8].

Meanwhile, logic programming has also been an important tool for knowledge base building, which

provides a declarative way for problem specification, well-founded semantics for formal reasoning, and non-deterministic mechanism for solution searching. However, research on combining all together logic programming, object-oriented programming and uncertainty logic appears to be sporadic.

The benefit of such a combination is not only that the advantages of the three disciplines can be exploited, but also that one discipline sheds light on existing problems resulting from combining the other two. In particular, those are ones of multiple inheritance and uncertain inheritance, which have not been adequately solved in previous works, as reviewed in [1], on extending the conventional object-oriented model to the uncertain object-oriented model.

The work on Fril++ ([4]), extending Fril ([3]) with object-oriented features, is among a few attempts towards uncertain object-oriented logic programming. However, that work still lacked a coherent framework for the implementation of Fril++ to be completed. Thus, in [1], we have proposed a general framework for object-oriented extension of existing fuzzy logic programming systems. Two schemes, namely, *probability-based* and *possibility-based*, have been provided in the framework, intended for fuzzy logic programming systems with formulas weighted by probability-based degrees or possibility-based degrees, respectively.

This paper applies the proposed probability-based scheme to the implementation of Fril++ by writing a translator to convert a Fril++ source program to a Fril target program, following the approach of [12]. Section 2 presents an overview of a Fril++ system and the translation process. Section 3 introduces

---

<sup>1</sup> Corresponding author.

Fril++ syntax and Section 4 discusses related semantic issues. Then Section 5 presents our translation strategy and illustrates translated codes. Finally, Section 6 presents concluding remarks of the paper.

## 2 Overview

### 2.1 General Class Hierarchy

Like any other object-oriented system, a Fril++ system is associated with a class hierarchy. Besides particular classes for the domain of the system, there are two special classes, namely, *Fril* and *Fril++*, which are common to all Fril++ systems. The *Fril* class is at the top of a class hierarchy, containing all Fril built-in predicates, which can be inherited by all classes in a Fril++ system. The *Fril++* class is a subclass of the *Fril* class, but a super-class of any other class, to define new common predicates for all Fril++ systems and to override some Fril built-in predicates. The predicates in the *Fril* and *Fril++* classes can also be overridden.

As discussed in [1], for a class hierarchy in an uncertain object-oriented system, we assume that a class is totally subsumed by any of its super-classes or, in other words, a class totally subsumes any of its subclasses. The first reason is that a set of classes with a graded inclusion or inheritance relation actually forms a *network* rather than a *hierarchy*, because if a class *A* has some inclusion degree to a class *B* based on a fuzzy matching of their descriptions, then *B* usually also has some inclusion degree to *A*. Second, naturally, a concept is usually classified into sub-concepts that are totally subsumed by it, though the sub-concepts can overlap each other.

### 2.2 Translation Process

As shown in [12], object-oriented logic programs can be translated into normal logic programs, such as Prolog ones, to be executed almost as efficiently as the original programs would be. We follow this approach in the implementation of Fril++ by writing a translator, using Fril itself, to convert Fril++ programs to Fril programs. Fril is a logic programming language that can handle both fuzzy and probabilistic uncertainties and, unlike other Prolog-like languages, Fril's syntax is Lisp-like with list as the primary data and program structure. A Fril atom, i.e., predicate, has the following form:

*(predicate-name arg1 arg2 ... argN)*

and the form of a Fril clause is as follows:

*(head-atom bd-atom1 bd-atom2 ... bd-atomN)*

A Fril++ source program, which contains class definitions and logical clauses, has the same list format as a Fril program. Before being translated into its Fril target program, it is loaded into system memory and converted to Fril internal format, from which the translator generates its Fril target program. This loading also performs lexical analysis on the Fril++ source program.

## 3 Fril++ Syntax

### 3.1 Class Definition

A Fril++ *class* definition contains the following sections: *super-class* declaration, *constant* declaration, *part* declaration, *attribute* definition, and *method* definition. The super-class section declares the *immediate* super-classes of the class. The constant section declares the constant labels and their values associated with the class, which can be inherited or overridden like class attributes and methods. The part section declares the identifiers and classes of the objects to be included as parts of an instance of the class.

The attribute and method sections define the *properties* of the class which, in the logic-based object-oriented model, are represented by logical clauses. Each attribute is associated with a *range* of values, defined on a domain, that the attribute can take. Generally, for the case with uncertainty, we assume an attribute value range to be a fuzzy set interpreted as a possibility distribution on its domain. Further, as in the probability-based scheme of the framework proposed in [1], each class property is associated with a support pair defining the lower and upper bounds of the probability for the property being applicable to a member of the class.

For examples, let us start with a simple one of the class BIRD defined in Fril++ as follows:

```
((class bird extends (winged-animal))
  (attributes ((no-of-legs 2)
               ((fly)) : (0.9 0.95)))
```

Here, words printed in boldface are Fril++ reserved words. For the attribute *fly*, 0.9 and 0.95 are respectively a lower bound and an upper bound of the conditional probability of an object being able to fly given that it is a bird.

For another example, one can have the following definition of the class MAN:

```
((class man extends (person))
  (parts (left_eye human_eye)
          (right_eye human_eye))
```

```

(attributes ((age [0:1 120:1]))
           (height [0:1 2.4:1]))
(methods ((handsome) (age young)
         (height tall)))

```

Here, [0:1 120:1] is a special fuzzy set, which is equivalent to a crisp one, defining the age of a man to be in between 0 and 120 years of age. Similarly, [0:1 2.4:1] defines the height of a man to be in between 0 and 2.4 metres.

The method is a rule saying “A man is handsome if he is *young* and *tall*”, where *young* and *tall* are linguistic labels of fuzzy sets. As the concepts *young* and *tall* are context-dependent, particularly on sex and geographical region, the fuzzy sets defining them depends on the class of the object that the method is applied to. For example, in the class ASIAN\_MAN they can be defined as follows:

```

((class asian_man extends (asian_person man))
 (constants (young [0:1 30:1 40:0 120:0]
             (tall [0:0 1.5:0 1.7:1 2.4:1])) ....)

```

where [0:1 30:1 40:0 120:0] represents a fuzzy set on [0, 120] whose membership function takes value 1 on [0, 30], value 0 on [40, 120], and is linearly decreasing on [30, 40]. Meanwhile, the membership function of [0:0 1.5:0 1.7:1 2.4:1] is linearly increasing on [1.5, 1.7].

### 3.2 Message Passing

The main issue of message passing is to identify where the attribute or the method referred to by the message is defined. It depends on a message’s receiver, whose “address” can be specified by either one of the following:

- The Fril++ reserved word **self**
- The Fril++ reserved word **super**
- A class identifier
- An object identifier, possibly followed by a sequence of part identifiers
- An object variable, possibly followed by a sequence of part identifiers.

The address **self** means that the receiver is the currently referred object. The address **super** means that the receiver is a super-class of the currently referred object. If the address is an identifier of a class, then the message is sent directly to that class. Meanwhile, if the address is an identifier or a variable representing an object, then the receiver is that object itself. If the address ends with an identifier of a part, then the message is sent to the corresponding object of that part. In the case when the receiver is an object, attributes or methods to be invoked are ones defined for the object itself or ones

that the object inherits from classes. Such an inheritance is possibly associated with an uncertainty degree, which will be discussed in Section 4.

For example, with the classes MAN and ASIAN\_MAN defined above, suppose that Lee belongs to ASIAN\_MAN. Then the following Fril++ queries respectively asks about the colour of Lee’s left eye and the support for Lee being a handsome man:

```

? ((lee.left_eye.colour X))
qs ((lee.handsome))

```

where ? and qs are query predicates built into Fril++.

## 4 Semantic Issues

### 4.1 Uncertain Membership Evaluation

In the uncertain object-oriented model, the membership of an object to a class is not a matter of “to be or not to be” as in the classical model, but rather a matter of degree. Here, for the probability-based scheme of the framework proposed in [1], uncertain membership is measured by support pairs.

It seems that there is no universally applicable method for evaluating the membership degree of an object to a class on the basis of their fuzzy and uncertain descriptions. Therefore, for flexibility, we propose that each class has, or inherits from its super-classes, a user-defined method for computing the membership degree of an object to the class. Nevertheless, as discussed in [1], there are good default definitions for such an uncertain membership evaluation method, which are *hierarchy-based* or *description-based*.

The hierarchy-based evaluation relies on the following assumptions:

1. If an object is a member of a class with some *positive characteristic* degree, then it is a member of any super-class of that class with the same degree.
2. If an object is a member of a class with some *negative characteristic* degree, then it is a member of any subclass of that class with the same degree.

These assumptions are a generalisation of ones stated in [7] and applied in [6] for fuzzy truth degrees, where the positive and negative characteristics were defined to be **true** and **false** characteristics, respectively. Here, probability lower bounds are considered as positive characteristic degrees, while probability upper bounds are

considered as negative characteristic ones. Consequently, if an object is a member of a class with a support pair  $(l\ u)$ , then it is a member of any super-class of that class with the support pair  $(l\ 1)$ , and a member of any subclass of that class with the support pair  $(0\ u)$ .

Meanwhile, the description-based evaluation relies on class attribute ranges, object attribute values and attribute applicability degrees. This is actually a weighted fuzzy pattern matching problem. Let, for every  $i$  from 1 to  $n$ ,  $R_i$  be the range of attribute  $A_i$  of a class  $C$ ,  $V_i$  be the value of  $A_i$  of an object  $O$ , and  $(l_i\ u_i)$  be the support pair for the applicability of  $A_i$  to  $C$ . Then, on the basis of the evidential logic matching method of [3], the support pair for  $O$  being a member of  $C$  is defined to be  $(F(\bigwedge_{i=1,n}\{w_i\ i\}))\ F(\bigwedge_{i=1,n}\{w_i\ i\})$ , where:

- $w_i = (l_i + u_i) / \bigwedge_{i=1,n}\{(l_i + u_i)\}$ , which measures a weight of  $A_i$  in evaluation of the membership of  $O$  to  $C$ .
- $(\ i\ i)$  is the support pair obtained by the semantic unification of  $R_i$  to  $V_i$ , based on mass assignment theory.
- $F: [0, 1] \times [0, 1] \rightarrow [0, 1]$  is a function whose choice determines the nature of the combination of  $A_1, A_2, \dots, A_n$ .

## 4.2 Uncertain Inheritance

In the classical object-oriented model, without exceptions, a class fully inherits all the properties of its super-classes and thus an object certainly has all properties of the classes to which it is a member. In the uncertain object-oriented model, due to uncertain membership of an object and uncertain applicability of a property to a class, inheritance naturally becomes uncertain. However, as reviewed in [1], this problem has not been adequately solved in previous works.

It was mainly due to the procedural interpretation of methods, as implicitly assumed in those works, for which it was not clear what an execution of a method with some uncertainty degree could mean. From the logic programming viewpoint, it simply means that the rule representing the method is weighted and applied with that uncertainty degree, just as in an uncertainty logic program. This viewpoint was applied and exemplified in [4], which however did not consider uncertain applicability of properties to a class.

We now derive the support pair for a property  $p$  of a class  $C$  being applicable to an object  $O$ . Let  $(l\ u)$  be the applicability degree of  $p$  to  $C$ , and  $(\ )$

be the membership degree of  $O$  to  $C$ . Also, we write  $p(x)$  and  $p(O)$  to denote “ $p$  is applicable to  $x$ ” and “ $p$  is applicable to  $O$ ”, and  $C(x)$  and  $C(O)$  to denote “ $x$  is a member of  $C$ ” and “ $O$  is a member of  $C$ ”, respectively. Then, applying Jeffrey’s rule ([11]), the probability for  $p(O)$  being **true** is defined by:

$$\Pr(p(O)) = \Pr(p(x) \mid C(x)) \times \Pr(C(O)) + \Pr(p(x) \mid \neg C(x)) \times \Pr(\neg C(O))$$

Here,  $l = \Pr(p(x) \mid C(x))$ ,  $u = \Pr(C(O))$ , and  $\Pr(\neg C(O)) = 1 - \Pr(C(O))$ .

Assuming that  $\Pr(p(x) \mid \neg C(x))$  is totally unknown, i.e., only  $0 \leq \Pr(p(x) \mid \neg C(x)) \leq 1$  is known, one obtains:

$$l \times \Pr(p(O)) \leq u \times (l + (1 - l))$$

whence the support pair for  $p$  being applicable to  $O$  is:

$$(l \times \min\{1, u \times (l + (1 - l))\}) \quad (4.1)$$

This result satisfies intuition in the following special cases:

- $(\ ) = (1\ 1)$ :  $l = \Pr(p(O)) = u$ . That is, if  $O$  is a full member of  $C$ , then  $O$  inherits  $p$  from  $C$  with the same support pair as defined in  $C$ .
- $(\ ) = (0\ 0)$ :  $0 = \Pr(p(O)) = 1 - u$ . That is, if  $O$  is certainly not a member of  $C$  then, from only this source of information, nothing can be said about the applicability of  $p$  to  $O$ .
- $(l\ u) = (1\ 1)$ :  $\Pr(p(O)) = 1$ . Here, as  $O$  can inherit  $p$  from other classes, the upper bound of  $\Pr(p(O))$  is not limited by the upper bound of  $\Pr(C(O))$ .
- $(l\ u) = (0\ 0)$ :  $0 = \Pr(p(O)) = (1 - u)$ . Here, if  $O$  is not a full member of  $C$ , i.e.,  $u < 1$ , then the fact  $p$  is not applicable to  $C$  does not entail that  $p$  is not applicable to  $O$ .

We note that in general the lower and upper bounds of  $\Pr(p(O))$  also depend on  $l$ , but not in this case as shown by Formula (4.1) when  $\Pr(p(x) \mid \neg C(x))$  is totally unknown.

For example, suppose that the class PENGUIN is defined as follows:

```
((class penguin extends (bird))
  (attributes ((fly)) : (0 0)) ... )
```

and Penny is a member of PENGUIN but Billie is not, that is, the support pairs for Penny and Billie being members of PENGUIN are  $(1\ 1)$  and  $(0\ 0)$ , respectively. Then, by Formula (4.1), the support pairs for the attribute *fly* being applicable to Penny and Billie are respectively  $(0\ 0)$  and  $(0\ 1)$ . In other words, one can infer that Penny cannot fly but cannot say anything about whether Billie can fly or not, from only this source of information.

### 4.3 Multiple and Overriding Inheritance

For multiple inheritance, the procedural interpretation of methods also raises the problem of deciding which method among those inherited of the same name is to be selected for execution. Meanwhile, from the logic programming viewpoint, that selection problem disappears, as discussed and exemplified in [12] on logic-based object-oriented programming and in [4] on its extension with uncertainty.

Indeed, in the logic-based object-oriented model, each object is viewed as a knowledge base of clauses representing all the properties that the object can have. Then inherited properties of the same name are included in the object's knowledge base just as clauses whose heads are predicates of the same name, which are usual in logic programming. A message sent to the object is viewed as a query on the object's knowledge base, which is answered by a Prolog style proof procedure. There can be different and, moreover, possibly conflicting answers to the query, and it is a role of uncertainty logic to resolve the conflict and combine the answers. The particular solution depends on the uncertainty logic theorem prover being applied.

For an example adapted from [2], suppose that there is a shape which resembles both a circle and a square, as illustrated in Figure 1, and one wants to evaluate its area approximately. The problem can be solved by considering the shape as a partial member of both the class CIRCLE and the class SQUARE with support pairs, for instance, (0.7 0.8) and (0.6 0.7) respectively.

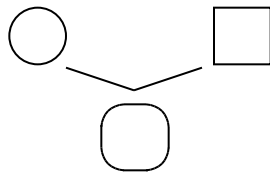


Figure 1: A shape resembling a circle and a square

The classes CIRCLE and SQUARE can be defined as subclasses of the class SHAPE as follows:

```
((class circle extends (shape))
  (attributes ((radius _ )))
  (methods ((area A)
            (radius R) (times R R R2)
            (times 3.14 R2 A))))
((class square extends (shape))
  (attributes ((length _ )))
  (methods ((area A)
            (length L) (times L L A))))
```

where  $A$ ,  $R$  and  $L$  are variables whose values can be fuzzy numbers, i.e., fuzzy sets on the set of all real numbers, and *times* is a Fril built-in predicate generally for fuzzy multiplication.

Then, the knowledge base of the shape virtually contains the above rules for evaluating its area, which are inherited from CIRCLE and SQUARE, respectively with the support pairs (0.7 1) and (0.6 1) as computed by Formula (4.1). As shown in [1] using Fril, with approximate radius and length assigned to the shape, both of the rules are applied and the two resulting support pair-weighted fuzzy sets can be combined into an expected fuzzy set, which can then be defuzzified to give an approximately estimated area of the shape.

Further, with overriding inheritance, it is possible for an object or a class not to inherit some properties from its super-classes, but use their own definitions of those properties, as in the classical object-oriented model. In Fril++, we adopt overriding inheritance as a default, that is, a property (possibly associated with a support pair) in a class is assumed to override properties of the same name in super-classes of the class.

However, in the case with uncertainty, the uncertain membership of an object to a class raises a new issue regarding overriding inheritance. Specifically, if an object is not a full member of a class, then the question is whether a property that the object inherits from the class would override properties of the same name in super-classes of the class. In Fril++, we assume that overriding inheritance is effective only with the full membership, i.e., with the support pair (1 1).

For example, the attribute ((fly)) : (0 0) of Penny in the example above, which is inherited from the class PENGUIN, overrides the attribute ((fly)) : (0.9 0.95) in the class BIRD. Now, suppose that Polly is a bird, but is not certainly a penguin, with the support pair (0.6 0.8) for instance. Then, by Formula (4.1), Polly has the following support pairs for its attribute *fly*:

```
((fly)): (0.9 0.95)
((fly)): (0 0.4)
```

which are inherited from BIRD and PENGUIN, respectively. In this case, as Polly is not a full member of PENGUIN, there is no overriding inheritance, but multiple inheritance instead. As in Fril, normally the intersection rule is applied to combine support pairs, but Dempster's rule ([14]) can be applied instead for those obtained from sources of independent and possibly conflicting viewpoints. Here, Dempster's rule gives the

combined support pair (0.782609 0.826087) for the attribute *fly* of Polly.

## 5 Translation to Fril

### 5.1 Translation Strategy

Following [12], the execution of an object-oriented logic program is considered to have two phases, namely, the *label phase* and the *body phase*. In the label phase, the system determines which classes contain definitions of the property (i.e., attribute or method) that is currently applied. Then, once those classes have been determined, the system enters the body phase to execute the property as defined in the bodies of the classes.

Corresponding to these label phase and body phase are *label clauses* and *body clauses* of the target program, which is a normal logic program, translated from an object-oriented logic program. The label clauses provide entry points to the properties in a class body, and to realise inheritance such that if a definition of a property is not found in a class, then it will be searched in super-classes of the class. Meanwhile, the body clauses are the translation of the definitions of class properties.

In the uncertain object-oriented model, which is out of the scope of [12], uncertain membership of an object and uncertain applicability of a property to a class make the following important differences between the translation of an uncertain object-oriented logic program and that of a classical object-oriented logic program, for both of the label clauses and the body clauses:

- In the classical model, an object as an instance of a class can inherit properties only from that class or its super-classes. Whereas, since in the uncertain model an object can be a partial member of a class, it can inherit with uncertainty degrees properties from any class.
- In the classical model, a property of a class is fully applicable to an object as a member of the class. Whereas, in the uncertain model, this applicability can be uncertain and, moreover, an associated uncertainty degree is not determinable at translation-time as a membership degree of an object to a class can change at run-time.

Regarding the first difference, for the label clauses of the translation of a classical object-oriented logic program, when a property is applied to an object, its definition is first searched in the local knowledge base of the object. If it is found

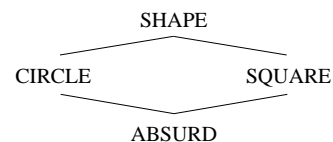
therein, then it is the definition to be executed. Otherwise, it is searched in the body of the class which the object is an instance of and then, if still not found, in the super-classes of that class. Meanwhile, in the uncertain case, every class containing a definition of the property is considered. Moreover, since a membership degree of an object to a class can change at run-time, overriding inheritance, which required the full membership as discussed in Section 4.3, is not determinable at translation-time.

Therefore, in Fril++, a definition of a property is first searched in the local knowledge base of the object that the property is applied to. If it is found therein, then it is the definition to be executed. Otherwise, not as in the classical case, it is searched in classes from the bottom of a class hierarchy of discourse. If it is found in a class, then its definition in the class is one to be executed. Moreover, if the object is a full member of the class, then the paths from the class to the top of the class hierarchy, but not the other paths, are pruned off the search. At the end, if there are more than one definition found in the search, one has multiple inheritance.

Regarding the second difference, the body clauses of the translation of an uncertain object-oriented logic program are weighted by support pair variables, whose values are determined at run-time. This can be realised for Fril++ as Fril allows variables to be support pairs of clauses. The next sections illustrate specifically how the presented strategy is realised in the translation of Fril++ class definition and message passing.

### 5.2 Label Clauses

For illustrating examples, let us use the following simple class hierarchy, excluding the default Fril and Fril++ classes introduced in Section 2.1, where ABSURD is just a dummy class to represent the class at the bottom of a class hierarchy, and CIRCLE and SQUARE are defined as in Section 4.3:



The super-class declaration in a class definition is translated to label clauses realising normal inheritance. For example, that of SQUARE is translated into the following Fril clauses:

```

((square PROP SELF) (square.super PROP SELF))
((square.super PROP SELF) (shape PROP SELF))
  
```

where *PROP* is a variable to represent a property, and *SELF* is a variable to represent, and pass around, an object currently referred to at run-time. The first clause says that, if a property *PROP* is not found in SQUARE, then look for it in the immediate super-classes of SQUARE which, in this example, include only SHAPE as stated by the second clause.

Such label clauses are preceded by those that provide the entry points to the properties defined in class bodies. For overriding inheritance, as discussed above, when a property of a class is applied to an object, first the membership of the object to the class is computed to determine whether overriding inheritance is effective, i.e., whether other definitions of the property in super-classes of the class are still applicable to the object. For example, the first entry point to the method *area* of SQUARE is defined by the following clause:

```
((square (area A) SELF)
  (!) (SELF (member square SUPP) SELF)
  (square. (area A) SELF SUPP))
```

Here, the *cut* predicate *!* is to deactivate the normal inheritance label clauses above. The support pair for the membership of *SELF* to SQUARE is computed by a method named *member*. This method can be defined in *SELF* itself, in SQUARE, or inherited from super-classes of SQUARE. As such, a search for its definition is carried out in the same way as those of other methods, starting at the local knowledge base of *SELF*. This explains the use of *SELF* as a variable predicate name, which is allowed in Fril, in the body of the clause.

Then *square.* is defined by the following clauses:

```
((square. (area A) SELF (1 1))
  (!) (square.area A SELF (1 1)))
((square. (area A) SELF SUPP)
  (square.area A SELF SUPP))
((square. PROP SELF _ )
  (square.super PROP SELF))
```

The first clause expresses that, if the support pair for a currently referred object, i.e., *SELF*, being a member of SQUARE is (1 1), then the method *area* of SQUARE is invoked. In this case, overriding inheritance is effective, which is realised by *!*, cutting paths to other definitions of *area* in super-classes of SQUARE as expressed by the third clause. Otherwise, as expressed by the second clause, this property of SQUARE is applied with the support pair for the membership of *SELF* to SQUARE, i.e., *SUPP*, passed along. In this second case, overriding inheritance is not effective, so *!* is not used.

### 5.3 Body Clauses

Each property definition in a class is translated into a body clause. For example, the body clause for the method *area* of SQUARE is as follows:

```
((square.area A SELF (X Y))
  (SELF (length L) SELF)
  (SELF (times L L A) SELF)
  (eq SUPP (X 1))) : SUPP
```

where *SUPP* defines the support pair for the applicability of the method *area* of SQUARE to *SELF*, in accordance to Formula (4.1). Here, the Fril built-in predicate *eq* is to equate *SUPP* with (X 1), as only simple support pair variables are allowed in Fril.

We note that, in the translation, a definition for *times* is also searched from *SELF* as for any other property, though in the given definition of *area* of SQUARE, *times* is intended to be the one built into Fril. To bypass classes in searching for a definition of a property, as mentioned in Section 3.2 for message passing, one can specify a class whose definition for the property is intended to be applied. For example, the definition of *area* of SQUARE can be modified as follows for this purpose:

```
((area A)
  (length L) (fril.times L L A))
```

Then its translation becomes:

```
((square.area A SELF (X Y))
  (SELF (length L) SELF)
  (times L L A)
  (eq SUPP (X 1))) : SUPP
```

in contrast to the previous translation.

In Fril++, clauses can be asserted to the local knowledge base of an object. We recall that, if a property is applied to an object, a definition for the property is first searched in the local knowledge base of the object and, if it is found therein, it is the only definition to be executed. As such, not as in the case of a class, the entry point to a property of an object and the translation of its definition can be grouped into one clause. For example, suppose the following Fril++ clauses for an object *ufo*:

```
((ufo.radius [1.0: 0 1.2: 1 1.4: 0]))
((ufo.length [1.8: 0 2.2: 1 2.6: 0]))
((ufo.member circle (0.7 0.8))
  (ufo.member square (0.6 0.7)))
```

where [1.0: 0 1.2: 1 1.4: 0] and [1.8: 0 2.2: 1 2.6: 0] are fuzzy sets defining the linguistic terms *about 1.2* and *about 2.2*, respectively. The clauses express that *ufo* is a member of CIRCLE and SQUARE with the support pairs (0.7 0.8) and (0.6 0.7), respectively, and has the radius of *about 1.2* and the length of *about 2.2*.

These clauses are translated into the following Fril clauses:

```
((ufo (radius [1.0: 0 1.2: 1 1.4: 0]) ufo) (!))
((ufo (length [1.8: 0 2.2: 1 2.6: 0]) ufo) (!))
((ufo (member circle (0.7 0.8)) ufo) (!))
((ufo (member square (0.6 0.7)) ufo) (!))
((ufo PROP ufo) (absurd PROP ufo))
```

Here, the last clause expresses that, if a definition of a property is not found in the local knowledge base of *ufo*, then it is searched in classes starting with ABSURD at the bottom of the class hierarchy, as discussed in Section 5.1. Meanwhile, the use of ! in the preceding clauses is to realise overriding inheritance.

Fril++ queries are also translated accordingly. For example, the following query asks for the area of *ufo*:

```
? ((ufo.area A))
```

and its translation is:

```
? ((ufo (area A) ufo))
```

Meanwhile, the translation for constants, with normal inheritance and overriding inheritance, is similar to that for attributes. The translation for message passing whose address is **self** or **super** is straightforward from its description in Section 3.2. The translation for message passing whose address is composed of an object identifier followed by a sequence of part identifiers is similar to the translation for message passing with an object identifier as a simple address.

## 6 Conclusion

We have presented Fril++ syntax and discussed related semantic issues in translating an uncertain object-oriented logic program to a normal uncertainty logic program. We have pointed out that, not as in the classical object-oriented model, in the uncertain model an object can inherit with uncertainty degrees properties from any class, and overriding inheritance depends on uncertain membership of an object to a class of discourse. Our translation strategy has then been presented and translated codes for Fril++ class definition and message passing have been illustrated. The presented translator has been implemented with the current version of Fril.

For a completed version of Fril++ to be released, there remain problems to be solved and their solutions to be implemented. Those include handling the access mode of a class, i.e., public, private, or protected, and that of a class property, i.e., public or private, and optimising translated codes. These are

among the topics that are currently investigated. Our direct application of Fril++ is then for building an uncertain object-oriented data browser as described in [2].

## References

- [1] Baldwin, J.F., Cao, T.H., Martin, T.P. and Rossiter, J.M. 2000. Towards soft computing object-oriented logic programming. In Proceedings of the 9<sup>th</sup> IEEE International Conference on Fuzzy Systems. To appear.
- [2] Baldwin, J.F. and Martin, T.P. 1995. Refining knowledge from uncertain relations - a fuzzy data browser based on fuzzy object-oriented programming in Fril. In Proceedings of the 4<sup>th</sup> IEEE International Conference on Fuzzy Systems, pp. 27-34.
- [3] Baldwin, J.F., Martin, T.P. and Pilsforth, B.W. 1995. *Fril - Fuzzy and Evidential Reasoning in Artificial Intelligence*. Research Studies Press.
- [4] Baldwin, J.F., Martin, T.P. and Vargas-Vera, M. 1998. Fril++: object-based extensions to Fril. In Martin, T.P. and Fontana, F.A. (eds), *Logic Programming and Soft Computing*, Research Studies Press, pp. 223-238.
- [5] Bordogna, G., Lucarella, D. and Pasi, G. 1994. A fuzzy object oriented data model. In Proceedings of the 3<sup>rd</sup> IEEE International Conference on Fuzzy Systems, pp. 313-318.
- [6] Cao, T.H. 1999. Foundations of order-sorted fuzzy set logic programming in predicate logic and conceptual graphs. PhD Thesis, University of Queensland.
- [7] Cao, T.H., Creasy, P.N. and Wuwongse, V. 1997. Fuzzy types and their lattices. In Proceedings of the 6<sup>th</sup> IEEE International Conference on Fuzzy Systems, pp. 805-812.
- [8] Dubitzky, W., Büchner, A.G., Hughes, J.G. and Bell, D.A. 1999. Towards concept-oriented databases. *Data & Knowledge Engineering*, 30, 23-55.
- [9] George, R., Buckles, B.P. and Petry, F.E. 1993. Modelling class hierarchies in the fuzzy object-oriented data model. *International Journal for Fuzzy Sets and Systems*, 60, 259-272.
- [10] Itzkovich, I. and Hawkes, L.W. 1994. Fuzzy extension of inheritance hierarchies. *International Journal of Intelligent Systems*, 62, 143-153.
- [11] Jeffrey, R. 1965. *The Logic of Decision*. McGraw-Hill.
- [12] McCabe, F.G. 1992. *Logic and Objects*. Prentice Hall.
- [13] Rossazza, J-P., Dubois, D. and Prade, H. 1997. A hierarchical model of fuzzy classes. In De Caluwe, R. (ed.), *Fuzzy and Uncertain Object-Oriented Databases: Concepts and Models*, World Scientific, pp. 21-61.
- [14] Shafer, G. 1976. *A Mathematical Theory of Evidence*. Princeton University Press.
- [15] Van Gysegheem, N. and De Caluwe, R. 1997. The UFO database model: dealing with imperfect information. In De Caluwe, R. (ed.), *Fuzzy and Uncertain Object-Oriented Databases: Concepts and Models*, World Scientific, pp. 123-185.