Formalizing fuzzy object role modeling schemas in the *FConQuer* system

**António Rosado**
Universidade do Algarve- FCT, DEEI
8005-139 Faro, Portugal
email: arosado@ualg.pt

**Rita A. Ribeiro**
UNINOVA
2825-114 Monte Caparica, Portugal
email: rar@uninova.pt

**Abstract**
The aim of this paper is to discuss the formulation of fuzzy object role modeling (FORM) schemas in *FConQuer*. The *FConQuer* system is a framework that allows to process fuzzy conceptual queries. To illustrate the FORM schema formulation we present some examples.

**Keywords:** Object-Role Modeling, Fuzzy Object-Role Modeling schema formulation, fuzzy conceptual queries.

**1 Introduction**
A Fuzzy Object Role Modeling (FORM) schema is an extension of the object-role modeling methodology (ORM) [10], where the value constraints can have fuzzy values. The concept of a FORM schema (see example in Figure 1) was first presented in [12] and in this paper we discuss its formalization.

Using the FORM formalization we developed the *FConQuer* framework, which is able to answer fuzzy queries over an object-oriented database management system (OODBMS).

Since the early 1990s research has been done [5] [7] [9] [13] on extending object-oriented data models (OODM), in order to develop OODBMSs capable of storing both imprecise and uncertain information. Our approach considers a crisp OODBMS and we only address imprecision in the queries. Further, we do not consider extensions of SQL-like query languages [6]. Specifically, we focus on the extension of a conceptual query language based on the ORM [10] to handle fuzzy queries.

To describe the formulation we use a soccer example [12] with data of the Portuguese Superleague championship for the first-half season of 2002-2003 [1]. In Figure 1 we depict a partial FORM, from this soccer example, that will be used to show some illustrative fuzzy conceptual queries.

**Figure 1:** FORM schema.

This paper is organized as follows. In section 2 we will briefly present the general architecture of the *FConQuer* framework. Section 3 explains the FORM schema formalization; Section 4 explains the formulation of the fuzzy extensions proposed for the ORM schemas. Section 5 explains how to create the FORM schema to formulate fuzzy queries using the *FConQuer* query conceptual language. In section 6 we discuss illustrative examples and we briefly describe the query processing in *FConQuer*. In section 7 we present the conclusions of this work.
2 FConQuer architecture

Figure 2 illustrates the architecture of the FConQuer system, which has two operating modes:

A. Mode supervisor (dashed box on the left). In this mode, the database administrator creates the conceptual schema (FORM schema) which automatically derives the logical schema (an Object Definition Language (ODL) [8] schema).

B. User mode (dashed box on the right). In this mode, the end-user formulates a FConQuer query which is translated into an object query language (OQL) query [8].

When the user formulates a query, the first task is to parse it. The parsing module verifies whether the query is a valid query, i.e., has the correct syntax. Here we assume that the formulated query is already a valid query.

Once a FORM schema has been defined, the module Mapping FORM schema to ODL schema converts it to an ODL schema, thus creating the logical schema of the application database. During the conversion, the module also generates some information, predicate graphs and mapping tables, that associate a given item in the FORM schema to the corresponding item in the ODL schema. That information is required for mapping a valid query to an object query language (OQL) query [8], which is performed by the module Mapping FConQuer query to OQL query.

As can be seen in Figure 2, the FConQuer system is an add-on system that executes on top of an OODBMS. It provides as output an OQL query, executed by the underlying OODBMS against an existing application database.

The two databases in Figure 2 (in gray and white) constitute the query context [11], that is, the required information to formulate a query and to retrieve the corresponding answer. Specifically, the query context consists of the FORM schema (the top database) and the application database (the bottom database). Hence, the query context is dependent on the application whereas the modules of the system do not depend on the application data. In reality, the independent modules represent the general framework proposed in this work.

The FConQuer framework has been implemented using the ObjectStore OODBMS (release 6.0 – December 2001), whose facilities are accessed via the Java interface to ObjectStore [2], a Java programming environment enhanced with classes provided by ObjectStore.

3 FORM formalization

Modeling fuzzy concepts such as good player and superstriker requires extending the ORM, specifically its value constraints [12]. In Figure 3, we depict the proposed FORM formalization that allows specifying fuzzy values. Fuzzy values are represented by associating a linguistic variable to a value constraint, through the fact type: ValueConstraint is associated to
LingVariable. For sake of clarity, the object type LingVariable, which models a linguistic variable, is defined in a separate diagram, presented in the next section.

![Diagram showing the ORM formalization of the ORM diagram in Figure 3.](image)

- **Each** EntityType is an ObjectType that is of OTkind ‘Entity’

a) The ORM formalization

The disjunctive mandatory role constraint on two of the Value Constraint’s roles specifies that a value constraint must have a value specification or be associated with a linguistic variable, where or includes also both cases. For example (see Figure 1), the value constraint on the object type Rate is associated with a linguistic variable, which includes fuzzy values such as low and high; and the value constraint on the object type Grade has a value specification, including integer values in the set {1…10} and is associated with a linguistic variable, denoted Grade, which includes fuzzy values such as bad and good.

Note that the fuzzy values in a value constraint of a FORM schema are not to be stored as values of a database attribute. In fact, as mentioned in section 1, our approach considers crisp databases. The fuzzy values in a FORM schema constitute the linguistic terms that can be specified in query formulation.

The proposed FORM formalization can be mapped into an ODL schema in order to implement it in an OODBMS. Figure 4 presents a possible ODL schema for that mapping.

Figure 4: An ODL schema for FORM schemas.

This ODL schema constitutes the main part of the database schema that we use to store the FORM schemas, such as the one depicted in Figure 1. The fuzzy extension to the ODL schema is detailed in the next section.

4 An ODL schema for linguistic variables

The ORM diagram in Figure 3 does not describe the linguistic variables for reasons of clarity. The linguistic variables are defined in the diagram of Figure 6, which for space reasons is presented as two diagrams in Figure 6 a) and Figure 6 b). The diagram can be obtained as follows.

Figure 5 presents the definition of linguistic variable AverageGrade (of a Player). This linguistic variable is expressed in terms of
values per match. This fact allows us to assert the fact type \textit{LingVariable is expressed in Unit}, which is depicted in Figure 6. Please note that we populated the formalization in Figure 6 with this instance (example in grey colour) for illustrative purposes.

![Graph of AverageGrade](image)

Figure 5: Linguistic variable \textit{AverageGrade}.

The universe of discourse (UoD) of the linguistic variable \textit{AverageGrade} is \([1,10]\) (see Figure 5). This set can be specified through a range, where value 1 is the lower limit and 10 is the highest limit (see Figure 6 a)). On this UoD we defined three fuzzy terms: \textit{bad}, \textit{average} and \textit{good}, identified by reference scheme numbers, ftNr: 1, 2, and 3 (Figure 6 a), top right). This allows asserting the fact type \textit{FuzzyTerm has FuzzyTermName}. Hence, an instance of this fact type is \textit{FuzzyTerm Nr 1 has FuzzyTermName \textquoteleft bad\textquoteright}, which is depicted in Figure 6 b) as the row (1, bad). Furthermore, we can also assert the fact type \textit{FuzzyTerm is defined by FuzzySet} from facts such as \textit{FuzzyTerm Nr 1 is defined by FuzzySet Nr 1} (see the row (1, 1) in Figure 6 b)).

There is an order relation among the fuzzy terms, specifically, \textit{bad < average < good}. This is modeled by assigning a position to each fuzzy term: the position numbers of fuzzy terms \textit{bad}, \textit{average}, and \textit{good} are 0, 1, and 2 respectively. These facts assert the fact type \textit{FuzzyTerm is in Position} (see Figure 6 b)).

The membership functions (fuzzy sets) defining the fuzzy terms \textit{bad}, \textit{average}, and \textit{good} are represented as follows. The fuzzy set defining the fuzzy term \textit{bad} (see Figure 5) is a trapezoidal membership function represented, by four elements \(x_i, \mu_{bad}(x_i)\): \([0.999,0), (1,1); (2,1); (4,0)]\). The ordered numbers 1, 2, 3 and 4, identify each element and are denoted by \textit{ElemNr} in Figure 6 b). Since we used trapezoidal functions we considered a small deviation on the domain to express the left and right trapezoids. With the four trapezoidal elements of \textit{bad} we assert the fact type \textit{FuzzySet holds Element} (see Figure 6 b)). For instance, the element \((2,1)\), with \textit{ElemNr} of 3, allows us to assert the fact types \textit{Element has Value} and \textit{Element has MembershipValue}, respectively 2 and 1. In Figure 6 b) this is represented by \textit{ElemNr} = 3 and \textit{Value} = 2; and \textit{ElemNr} = 3 and \textit{MembershipValue} = 1.

Figure 6: A FORM formalization for the \textit{LingVariable} object type with an instantiation.

Similarly, the fuzzy sets defining the fuzzy terms \textit{average} and \textit{good} (see Figure 5) have trapezoidal membership functions represented,
respectively, by \([2.0); (4.1); (6.1); (7.0)\] and \([5.0); (7.1); (10.1); (10.001,0)\]. For clarity reasons, in Figure 6 we do not populate the ORM diagram with the facts corresponding to fuzzy terms average and good.

Figure 7 presents the ODL schema for linguistic variables derived from the FORM formalization in Figure 6. Note that since the object type \textit{universe of discourse} (\textit{UoD}) in Figure 6 a), is described in a similar way to the object type \textit{value specification} (\textit{ValueSpec}) in Figure 3 b), it is mapped to the class \textit{ValueSpec} (see the attribute \textit{LingVariable:has} in Figure 7).

class LingVariable {
  attribute string lingVarName;
  attribute string unit;
  attribute ValueSpec has;
  relationship set<FuzzyTerm> takes;
  inverse FuzzyTerm:belongsTo;
}

class FuzzyTerm {
  attribute unsigned short fNr;
  attribute string fuzzyTermName;
  attribute unsigned short positionNr;
  relationship LingVariable belongsTo;
  inverse LingVariable:takes;
  relationship FuzzySet belongsTo;
  inverse FuzzySet:belongsTo;
}

class FuzzySet {
  attribute unsigned short fnNr;
  attribute string fuzzyTermName;
  attribute unsigned short elemNr;
  attribute double hasMembership;
  relationship FuzzySet belongsTo;
  inverse FuzzySet:belongsTo;
}

class StringElement {
  extends Element {
    attribute string has;
  }
}

class NumberElement {
  extends Element {
    attribute double has;
  }
}

Figure 7: ODL schema for linguistic variables.

The ODL schemas in Figure 4 and Figure 7, when considered together, allow storing a FORM schema in an OODBMS. In the next section, we will describe how the OODBMS administrator can create a FORM schema using the \textit{FConQuer} framework.

5 Creating a FORM schema

In sections 3 and 4 we specified the concept of FORM formalization using the ODL language. Hence, any particular FORM schema is a collection of objects, each of which belongs to a class in the \textit{UoD} specification. Therefore, designing a FORM schema corresponds to inserting objects into an object data base (ODB). In this section, we describe the functions we provide for storing the objects of a particular FORM schema into the ODB.

In [4], the authors implemented a user interface that allows the user to design the (fuzzy) schema of a database (fuzzy object-oriented database) in a similar way to which a user constructs a graph using the Microsoft Draw environment. That is, the user clicks on a button with the mouse to insert some shape into the graph and then, without user perception, some function is called to issue a sequence of commands, in a Data Definition Language, to perform the corresponding action on the database schema.

Since here we are only concerned with conceptual queries, we consider that when the user (a database administrator) wants to create a schema, he/she has three types of functions:

1. Insertion functions. Each of these allows adding an element (object type, role, predicate, constraint) to the schema. For example, when the user wants to insert an object type into the schema, he/she is asked to supply its name and kind (see the class \textit{ObjectType} in Figure 4). Note that the attribute \textit{plays}, which links the object type to its roles, is filled using the appropriate connection function. Moreover, when the user inserts most of the elements (predicates, roles, constraints, value specifications), he/she has to provide a number (see Figure 4) identifying that element.

2. List functions. Each function allows displaying all instances of a given type of element;

3. Connection functions. These make the necessary links between two elements in the schema. For example, there is a function to link a role to an object type. This function, which takes an Object-type object and a role object, adds the Role object to the set of roles the Object-type object plays. Note that the set of roles of an Object-type object is given by the attribute \textit{plays} of the Object-type object (see the class \textit{ObjectType} in Figure 4).

In the complete soccer example implementation, the FORM schema database has the following population: 15 object types, 48 roles, 20 predicates, 42 constraints, 2 value specifications, and 5 linguistic variables. Here we only discuss the subset relative to fact types of Player (Figure 1).

Note that the derived fact types are not stored in the FORM schema database. For example, in the derived fact type \textit{Player has PlayingRate} (see Figure 1), the object type \textit{PlayingRate}, its two
roles, its mandatory role and uniqueness constraints are all objects not included in the database. Moreover, the value specification in the value constraint on the object type AverageGrade is also not stored in the database.

Considering that we can’t derive the fuzzy values of a linguistic variable, from the base fact types, we have to store in the database the linguistic variables associated with derived fact types and its corresponding value constraint. For example, the linguistic variable PlayingRate (see Figure 1) and the value constraint on PlayingRate are stored in the database.

The stored FORM schema could be used by any user interface to guide the ordinary user to formulate a valid FConQuer query. This user interface was called, in Figure 2, query formulation module.

### 6 Illustrative queries for Soccer example

In table 1, we present the three main types of queries the FConQuer language can deal with and an example for each one.

<table>
<thead>
<tr>
<th>FConQuer Taxonomy</th>
<th>Query Formulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Queries with basic query path</td>
<td>Q1. Player scores with Rate = high</td>
</tr>
<tr>
<td>II. Queries with composed query path</td>
<td>Q2. Player got Grade = good in Match that has visitor Team = Porto</td>
</tr>
<tr>
<td>III. Tree-shaped queries</td>
<td>Q3. Player scores with Rate = high and Player has PlayingRate = low</td>
</tr>
</tbody>
</table>

A type-I query consists of just one predicate. A type-II query consists of two or more predicates, which have been connected to build a query path. That is, a type-II query consists of two or more connected type-I queries. A type-III query consists of two or more linear acyclic paths, which have been connected through a common object type to build a tree shaped path. That is, a type-III query consists of two or more connected type-II queries.

Now we present the results for the illustrative queries presented in Table 1.

**Q1. Which players are superstrikers?**

has the following formulation in FConQuer:

Player scores with Rate = high.

And the corresponding OQL expression:

```sql
SELECT p.playerNr, p.rateIsHigh( ) FROM Players p WHERE p.rateIsHigh( ) > 0 ORDER BY p.rateIsHigh( ) DESC
```

returns the following results:

<table>
<thead>
<tr>
<th>Name</th>
<th>Team</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adriano</td>
<td>Nacional</td>
<td>1.0</td>
</tr>
<tr>
<td>Ricardo</td>
<td>Beira-Mar</td>
<td>1.0</td>
</tr>
<tr>
<td>Beira-Mar</td>
<td>Antchouet</td>
<td>1.0</td>
</tr>
<tr>
<td>Laranjeiro</td>
<td>U.Leiria</td>
<td>1.0</td>
</tr>
<tr>
<td>Dario</td>
<td>Academica</td>
<td></td>
</tr>
<tr>
<td>Romeu</td>
<td>V.Guimaraes</td>
<td>1.0</td>
</tr>
<tr>
<td>Mantorras</td>
<td>Benfica</td>
<td>0.38</td>
</tr>
<tr>
<td>Paulo_Alves</td>
<td>Gil_Vicente</td>
<td>1.0</td>
</tr>
<tr>
<td>Joao_Pedro</td>
<td>Santa_Clara</td>
<td>0.38</td>
</tr>
<tr>
<td>Simao</td>
<td>Benfica</td>
<td>1.0</td>
</tr>
<tr>
<td>Joao_Pinto</td>
<td>Sporting</td>
<td>0.32</td>
</tr>
<tr>
<td>Gaucho</td>
<td>Maritimo</td>
<td>1.0</td>
</tr>
<tr>
<td>Fanguiero</td>
<td>V.Guimaraes</td>
<td>0.29</td>
</tr>
<tr>
<td>Ceara</td>
<td>Santa_Clara</td>
<td>0.85</td>
</tr>
<tr>
<td>Miguel</td>
<td>Setubal</td>
<td>0.17</td>
</tr>
<tr>
<td>Tiago</td>
<td>Benfica</td>
<td>0.83</td>
</tr>
<tr>
<td>Mauro</td>
<td>P.Ferreira</td>
<td>0.17</td>
</tr>
<tr>
<td>Postiga</td>
<td>FC_Porto</td>
<td>0.69</td>
</tr>
<tr>
<td>Cavaleiro</td>
<td>Moreirense</td>
<td>0.17</td>
</tr>
<tr>
<td>Deco</td>
<td>FC_Porto</td>
<td>0.69</td>
</tr>
<tr>
<td>Kibuey</td>
<td>U.Leiria</td>
<td>0.17</td>
</tr>
<tr>
<td>Manoel</td>
<td>Gil_Vicente</td>
<td>0.58</td>
</tr>
<tr>
<td>Jardel</td>
<td>Sporting</td>
<td>0.17</td>
</tr>
<tr>
<td>Barroso</td>
<td>Sp.Braga</td>
<td>0.56</td>
</tr>
</tbody>
</table>

As we can see, only 25 players (from 478 existing players in the ODB) have a non-null membership degree. The first seven fit the category completely, i.e. they are superstrikers with membership 1. The others are not so good superstrikers since their membership degrees are lower.

**Q2. Which players got a good grade in a match played by Porto as a visitor team?**

has the following formulation in FConQuer:

Player got Grade = good in Match that has visitor Team = Porto.

And the corresponding OQL expression:

```sql
SELECT g.player.playerNr, g.gradeIsGood( ) FROM (SELECT m.player FROM Matches m WHERE m.visitorTeam.teamName = "Porto") m, m.grades g WHERE g.gradeIsGood( ) > 0 ORDER BY g.gradeIsGood( ) DESC
```

returns the following results:

<table>
<thead>
<tr>
<th>Name</th>
<th>Team</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deco</td>
<td>FC_Porto</td>
<td>1.0</td>
</tr>
<tr>
<td>Carvalho</td>
<td>FC_Porto</td>
<td>1.0</td>
</tr>
<tr>
<td>Costa</td>
<td>FC_Porto</td>
<td>1.0</td>
</tr>
<tr>
<td>Postiga</td>
<td>FC_Porto</td>
<td>1.0</td>
</tr>
</tbody>
</table>
The results include 109 items: 42 with membership degree equal to 1.0 and 67 with membership degree equal to 0.5. For space reasons, we only show a subset.

Note that the results above show the importance of incorporating quantifiers in the FConQuer language. In fact, some players (such as Deco) appear several times in the answer. The reason is that there are about height matches in which Porto played as a visitor team (considering a half season). Hence, retrieving players with a good grade in those matches considers the players that got a good grade in each match. If we had applied the universal quantifier to Match, the query would have retrieved the players that had got a good grade in all such matches. Hence, a player would have appeared in the answer at most one time.

Q3. Which superstriker(s) is a bench warmer?

has the following formulation in FConQuer:

Player scores with Rate = high and Player has PlayingRate = low.

And the corresponding OQL expression:

```
SELECT Struct(field1: q.field1

    membership: MIN(p.membership,
                               q.membership))
FROM (SELECT p.playerNr,
            p.playingRateIsLow( )
FROM Players p
WHERE p.playingRateIsLow( ) > 0
ORDER BY p.playingRateIsLow( )
DESC) p,
(SELECT p.playerNr,
    p.rateIsHigh( )
FROM Players p
WHERE p.rateIsHigh( ) > 0
ORDER BY p.rateIsHigh( ) DESC) q
WHERE p.field1 = q.field1
```

returns the following results:

- Laranjeiro U.Leiria 1.0
- Cavaleiro Moreirense 0.17
- Kibuey U.Leiria 0.17

As we can see, only 3 players (from 478 players in the database) can be considered bench warmers (i.e. rarely called to play) and only one with absolute certainty (Laranjeiro, membership 1). This is quite logical because coaches tend to use extensively superstriker players.

In summary, the results obtained show that:

1. The FConQuer formulations are close to natural language, thus making FConQuer an interesting conceptual query language for users.
2. By contrast the OQL formulations are very different from natural language thus showing that OQL is not user-friendly for users.
3. Moreover, as the complexity of the queries increases, the formulations remain easy to formulate whereas the OQL formulations reach very complex formulations.

These facts illustrate the benefits of the approach used in this work and corroborate the claims of the crisp version by Halpin [3] and Owei [11]. Specifically, the idea behind this work is to provide the user with a conceptual query language and let the system do the mapping to a logical query language. In addition, this work proposed an extension to the crisp ORM approach by allowing fuzzy terms in the conceptual query language, thus allowing the possibility of using human-like imprecise terminology in the queries such as good grade, superstriker.

6.1 Query processing

As we mentioned in section 2, the modules that perform the query processing will be described in detail in a future paper. However, in this section, we briefly explain how the FConQuer system calculates the matching degrees in query results.

The module mapping FORM schema to ODL schema creates an ODL schema corresponding to the given FORM schema. In this mapping, each fuzzy term of a value constraint maps to a method, denoted fuzzy method, which belongs to a class in the generated ODL schema. For example, the fuzzy term high in the value constraint on the value type Rate (see Figure 1) maps to the fuzzy method rateIsHigh( ), which belongs to class Player.

A fuzzy method first calculates the crisp value x, which is the value of the corresponding value type in the FORM schema. For example,
rateIsHigh( ) first calculates the rate of a player (this involves dividing the number of goals scored by a player by the number of matches in which he played). Then, it calculates $\mu_{\text{fuzzy term}}(x)$. For example, rateIsHigh( ) calculates $\mu_{\text{high}}(x)$, where high is the fuzzy set that represents the linguistic term high of the Rate linguistic variable.

Note that in the first example in section 6, the fuzzy method rateIsHigh( ) is used in the SELECT clause of the OQL query for displaying each matching degree in the query results.

7 Conclusions

We described a fuzzy extension for the object-role modeling (FORM) which allows processing Fuzzy ConQuer (FConQuer) queries. Specifically, we explained in detail the modules of the FConQuer system that allow the user to construct a FORM schema and then perform conceptual queries.

Finally, we presented the results of executing some quasi-natural queries, using FConQuer formulation and generating OQL queries on the underlying OODBMS, using a soccer application database, which has been previously populated.

This work illustrates the suitability and usefulness of this general approach, i.e. the use of fuzzy conceptual query languages for querying ODBs, by using a new layer built on top of the logical layer provided by the standard OQL language.

As future work we intend to extend the FConQuer query language, thus allowing more complex queries. In particular, would be interesting to study the introduction of (fuzzy) quantifiers. Also, we intend to address in detail the mapping of a FORM schema to an ODL schema and the mapping of a FConQuer query to a OQL query. Finally, we plan to test the performance of FConquer.

References