COQUEL: A CONCEPTUAL QUERY LANGUAGE BASED ON THE ENTITY-RELATIONSHIP MODEL

Rafael Bello and Jorge Lloret

Department of Computer Science, University of Zaragoza, Spain
rafabellof@outlook.com and jlloret@unizar.es

ABSTRACT

As more and more collections of data are available on the Internet, end users but not experts in Computer Science demand easy solutions for retrieving data from these collections. A good solution for these users is the conceptual query languages, which facilitate the composition of queries by means of a graphical interface. In this paper, we present (1) CoQueL, a conceptual query language specified on E/R models and (2) a translation architecture for translating CoQueL queries into languages such as XQuery or SQL..

KEYWORDS

Conceptual Query Language, SQL, final user

1. INTRODUCTION

As brilliantly explained in [1], database systems are difficult to use due to a set of five pain points including (1) the cognitive load of learning the concepts of a query language and (2) the need to deal with implementation issues of the underlying database.

Moreover, with the spread of the web, more and more collections of data are becoming available to everyone in fields from biology to economy or geography. End users, but not experts in Computer Science, demand easy ways to retrieve data from these collections.

In an effort to simplify the query of databases, in this paper we propose to add a conceptual layer on which the user can specify the queries. For this layer, we have chosen the Entity/Relationship model (E/R model for short) because it is widely recognized as a tool which facilitates communications with end users and we would strongly argue that it also facilitates query writing. To put this into practice, we propose a new architecture which integrates: (1) the CoQueL language, which allows us to specify conceptual queries on an E/R model (2) a graphical interface built on the CoQueL language and (3) a translation from the graphical query to a target language such as XQuery or SQL.

The advance of this paper with respect to other works is twofold. First, it is thought to query data in several formats as relational or XML from a conceptual level. Second, we have gathered the best recommendations about visual queries and we have integrated them into our interface.

There have been several papers in the literature about query languages for end users. Some of the papers such as QBE [2] or spreadsheet algebra [3] lack a conceptual level unlike our paper which includes it because it facilitates the specification of queries. Other papers such as SQBE [4] or the query system for NeuronBank [5] are intended for a particular format of data unlike our paper which is intended for formats such as relational or XML. The paper QueryViz [6] does the reverse work because it generates conceptual queries from SQL queries. The paper CQL [7] also includes a conceptual level and we have borrowed from it concepts such as query abbreviation. However, its interfaces are a bit cluttered, so we offer a more simplified interface.

The rest of the paper is organized as follows. In Section 2, we explain the E/R metamodel and the relational metamodel as well as a running example. In Section 3, we introduce the CoQueL query language. Section 4 describes the query architecture and Section 5 its implementation. In Section 6, the graphical interface is presented. Moreover, in Section 7 we detail the related work and in Section 8 we show the conclusions and future work.

2. METAMODELS

CoQueL queries are specified in a context consisting of an E/R model, the corresponding relational model and a physical model populated with data in a particular RDBMS against which the queries are executed. So, let us explain the metamodels for building E/R models and relational models.

2.1 Entity/Relationship Metamodel

For building E/R models, we will use a conceptual metamodel based on the model proposed by Chen in [8].

According with the metamodel, an E/R model includes entity types, which are described by means of attributes. An attribute only describes one entity type and an entity type is described by one or more attributes.We can establish one or more relationship types between two entity types. Each entity type participates in a relationship type in the first position or in the second position. Each entity type participates in a relationship type with a cardinality, which denotes the number of instances of the other participant with which an instance of the initial participant can be related. The possible cardinality values are 0-1, 0-N, 1-1, 1-N where the first(second) value indicates the minimum(maximum) cardinality.

There are some integrity constraints associated to the metamodel: (ic1) two distinct entity types must have different names, (ic2) two distinct relationship types must have different links and (ic2) the names of the entity types and the links of the relationship types must be different.

In this paper, we will use as an example the E/R model of Figure 1, about the employees of an entreprise.

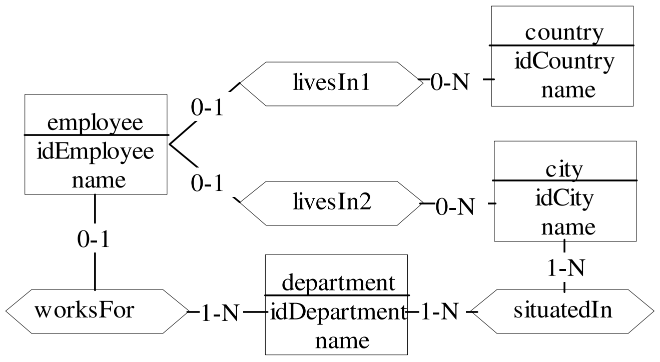


Figure 1. First example of an E/R model

2.2 Relational Metamodel

For building relational models, we will use the relational metamodel proposed by Codd in [9].
A relational model includes tables, which are described by means of columns. A column describes only one table and a table is described by means of two or more columns. Each table has a name. Each column has a name and a datatype. Each table has a primary key, which is formed by exactly one column. There are some integrity constraints associated to the metamodel: (ic4) two distinct tables must have different names, (ic5) two distinct columns of the same table must have different names. For translating an E/R model to a relational model there are, basically, two options [10]. The first one consists of translating every relationship type into a table. The second one translates the relationship types depending on the cardinality of the participants. When the relationship type is 0-N or 1-N for both participants, it is translated into a table; otherwise, it is translated into a foreign key. In this paper, we will follow the latter option. So, for the E/R model of Figure 1 the corresponding relational model is:
employee(idEmployee, name, idCountry, idCity, idDepartment)

country(idCountry, name)

city(idCity, name)

department(idDepartment, name)

situatedIn(idDepartment, idCity)

3. COQUEL LANGUAGE

The CoQueL language allows us to specify three kinds of conceptual queries on E/R models: linear, star and mixed. The linear queries are those which linearly traverse the E/R model. The star queries includes a root entity type and several relationship types whose common participant is the root entity type. The mixed queries are combinations of linear queries and star queries. For formalizing these intuitive ideas, we first define our notion of path, next the notion of CoQueL query and finally we show some examples of CoQueL queries.

3.1 Path

For formalizing the notion of CoQueL query, we previously introduce several notions of paths defined on E/R models. We present some examples of paths on the E/R model of Figure 2, where for simplicity we have omitted the attributes.

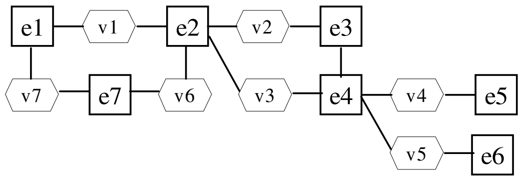


Figure 2. Second example of an E/R model

Definition 1 A lpath is an expression of the form

e1v1e2v2 . . . envn

where

n >= 1

ei i=1. . . n is an entity type

vi is the link of a relationship type between ei and ei+1

en is an entity type or an spath. If n=1, then e1 is an entity type

When en is an entity type, the lpath is called basic lpath.

Definition 2 A spath is an expression of the form

e(&(v1f1)&(v2f2) . . . &(vmfm))

where

m >= 2

vi is the link of a relationship type between e and fi i=1. . . m

fi is a lpath or a spath

When every fi is an entity type, the spath is called basic spath. An arm of the path is a linear path of the form evifi

Definition 3 A qpath is an lpath or an spath

Next we show examples of paths on the E/R model of Figure 2.

Three examples of basic lpath are: e1 e1v1e2 e1v1e2v2e3

An example of basic spath is e2(&(v2e3)&(v3e4))

Two examples of qpaths are e2(&(v2e3)&(v3e4(&(v4e5)&(v5e6))))

e1v1e2(&(v2e3)&(v3e4(&(v4e5)&(v5e6))))

3.2 CoQueL Query

A CoQueL query is a qpath with extra information for the entity types of the qpath. A complete specification of an entity type of a qpath consists of four clauses as follows: `entityTypeName(attributes; attributesCondition; attributesGroup; groupCondition)`

where attributes are the attributes which take part of the result of the query, attributesCondition is a conditional expression on the attributes of the entity type, attributesGroup are the attributes through which the instances of the entity type are grouped and groupCondition is a conditional expression on the groups indicated by means of attributesGroup.

Let us explain briefly each component.

Attributes The clause attributes indicates the selected attributes of the entity type to be displayed in the result. If we want to select all the attributes of the entity type, we can write only an asterisk instead of the name of all the attributes. It there is no condition on the attributes of the entity type and we do not want to select any of these attributes, we write the name of the entity type without parentheses. Let us see some examples. $e1$ means that we need the entity type $e1$ for the query but we do not select any attribute of this entity type and we do not impose any condition on these attributes. $e1(*)$ means that all the attributes of $e1$ are selected but we do not impose any condition on them. $e1(a11)$ means that the attribute $a11$ is selected but we do not impose any condition on the attributes of $e1$. In this clause, we can also include aggregation functions on the attributes.

Conditions The conditions on attributes and on groups can be simple or compound. A simple condition is a comparison condition or a condition with the LIKE predicate. A compound condition is the union by means of AND, OR or NOT of conditions.

A comparison condition has the form expr1 op expr2 where expr1, expr2 are scalar expressions and op is a comparison operator. The comparison operators are the usual ones: = < > <> <= >=. For example, $e1(*;\ a11 = 1)$ means that all the attributes of $e1$ are selected but only for those instances of $e1$ for which the value of attribute $a11$ is 1. $e1(a12;\ a11 = 1)$ means that attribute $a12$ of $e1$ is selected but only for those instances of $e1$ for which the value of attribute $a11$ is 1.
A LIKE condition has the form expr1 LIKE pattern where expr1 is a string expression and pattern is a representation of a set of strings.

Scalar expression The scalar expressions can be numeric expressions or string expressions.
A numeric expression is an arithmetic expression which includes as primaries one or more of the following: attribute names, possibly qualified, or numeric literals or aggregate functions or numeric expressions enclosed between parentheses. The aggregate functions, used for elaborating statistics, are COUNT, SUM, MIN, MAX and AVG.

If the numeric expression is used in an attributesCondition, then the primaries are attribute names, possibly qualified, or numeric literals or numeric expressions enclosed between parentheses. If the numeric expression is used in a groupCondition, then the primaries are numeric literals or aggregate functions or numeric expressions enclosed between parentheses.

A string expression is an expression which includes as primaries one or more of the following: attribute names, possibly qualified or string literals or the concatenation of several of them.
The complete syntax of a CoQueL query can be seen in Appendix A. To date, the CoQueL queries are equivalent to SQL single-block query expressions as defined in [3].

3.3.Examples

Let us see some examples of queries on the E/R model of employees shown in Figure 1.
Query 1 (linear). Find the employees who work in the purchasing department in Zaragoza

```
employee(*) worksFor department(;name='Purchasing')
situatedIn city(;name='Zaragoza')
```

Query 2 (mixed). For each department, find its name, the name of its employees, the name of the city and the country where the employees live.

```
department(name) worksFor employee(name)
```

```
(& (livesIn1 country(name)) & (livesIn2 city(name)))
```

Query 3 (with group conditions). Find the name and identifier of the countries where more than five employees live.

```
employee(-;-;-;COUNT(idEmployee)>5) livesIn1

country(idCountry, name;-;idCountry,name;-)
```

4. ARCHITECTURE FOR TRANSLATING GRAPHIC QUERIES INTO TARGET QUERIES

One of the main purposes of this work is to facilitate querying databases for non-expert users. To this end, we have defined a generic architecture which translates graphical queries into queries in a target language(such as XQuery or SQL) by using the CoQueL language in an intermediate step. Also, the CoQueL queries can be stored to be retrieved later or to be exchanged between different systems.

The architecture (see Figure 3) has two components: the models component and the query component. The models component consists of three models: the E/R model, the logical model and the correspondence model. The latter stores the correspondence between the E/R elements and target query language expressions. It is generated by the model translator when the E/R model is translated into the logical model and by taking into account the translation rules applied to the E/R model.

The query component includes three modules: the text query generator, the query validator and the target query generator. It works as follows: the final user graphically builds a query, based on the E/R model, and sends it to the text query generator, which transforms it into a CoQueL query. This query is validated syntactically by the query validator module. If the query is wrong, a message is sent to the user informing about this fact. If the query is right, it is the input for the target query generator module. This module uses the correspondence between E/R elements and target expressions of the model component and produces the target query as output.

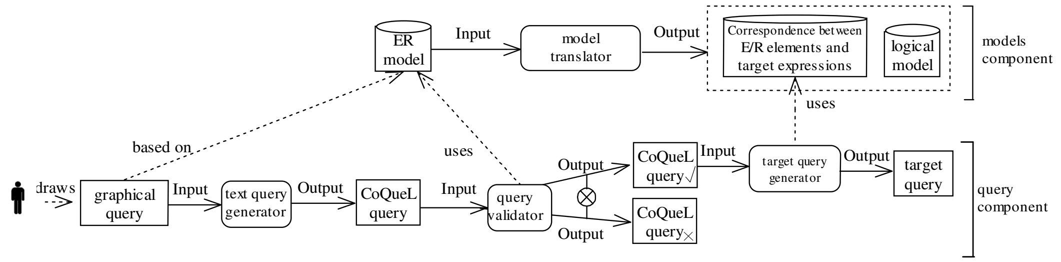


Fig. 3.Architecture for conceptual queries

5. AN IMPLEMENTATION OF THE ARCHITECTURE

Next, we specifiy how the models and algorithms of the architecture are implemented for SQL as target language.

5.1. Models Component Implementation

The E/R model is implemented as a relational database whose tables are entityType, relationshipType and attribute.

The correspondence between E/R elements and SQL expressions is implemented as a table called er2SQL. By SQL expressions, we refer to the following expressions which form part of a SQL query: table names, fully qualified column names and join conditions. The rules for generating the rows of the table er2SQL are as follows:

– For each entity type, the column table stores the name of the table into which the entity type is translated. The column expression is always null (see an example in row 1 of Table 1).

– For each relationship type, the column table stores the name of the table into which the entity type is translated or null if there is no such table. For example, if according to the translation rules, only the 0-N or 1-N relationship types are translated into a table, then the rest of the relationship types will have null value in the column. Moreover, the column expression stores

row	**conceptual element**	**table**	**expression**
1	employee	employee	null
2	livesIn1	null	employee.idCountry=country.idCountry
3	situatedIn	situatedIn	situatedIn.idDepartment=department.idDepartment AND situatedIn.idCity=city.idCity
4	idEmployee	null	employee.idEmployee

Table 1. Some rows of table er2SQL

5.2 Query Component Implementation

The text query generator, the query validator and the target query generator are implemented as the algorithms graphic2CoQueL, isValid, splitCoQueLQuery and writeTargetQuey. All of them are specified next.

```
Algorithm graphic2targetQueryLanguage(p language)

Input: Graphical specification of the query
Output: Query specified in the target query language
Pseudocode
repeat
coquelQuery← graphic2CoQueL()
until isvalid(coquelQuery)
(l entityTypes, l relationshipTypes, ll attributes ,l conditions,
ll groupAttributes, ll groupConditions) ← splitCoQuelQuery(coquelQuery)
writeTargetQuery (l entityTypes, l relationshipTypes, ll attributes,
l conditions, ll groupAttributes, ll groupConditions, p language)
```

```
Algorithm splitCoQueLQuery(q)

Input: CoQueL query
Output: Lists of entity types, relationship types, attributes, groups and conditions
```

```
involved in the query
Pseudocode
l entityTypes←getEntityTypes(q)
ll attributes←getAttributes(q,l entityTypes)
l relTypes← getRelTypes(q)
l conditions← getConditions(q)
ll groupAttributes← getGroupAttributes(q)
l groupConditions← getGroupConditions(q)
```

```
Algorithm writeTargetQuery (pl entityTypes, pl relationshipTypes, pll atttributes,
pl conditions, pll groupAttributes, pl groupConditions, p language)
INPUT: Lists of entity types, relationship types, attributes, groups and conditions
involved in the query
OUTPUT: target query
Pseudocode
IF p language='SQL' THEN
SELECTclause←buildClause('SELECT', pl entityTypes, pll atttributes)
FROMclause←buildClause('FROM', pl entityTypes, pl relTypes)
WHEREclause←buildClause('WHERE',pl entityTypes, pl relTypes,
pl conditions)
GROUPBYclause←buildClause('GROUP BY',pl entityTypes,
pll groupAttributes)
HAVINGclause←buildClause('HAVING',pl entityTypes, pl groupConditions)
SQLquery←buildSQL(SELECTclause, FROMclause, WHEREclause,
GROUPBYclause, HAVINGclause)
return SQLquery
END IF
```

The way of working is as follows: once the user has specified the graphical query, the graphic2CoQueL algorithm translates the graphical query into Co- QueL and it is validated until a correct CoQueL query is obtained. Then, the splitCoQueLQuery extracts the entity types, relationship types and attributes from the CoQueL queries into variables. Finally, the writeTargetQuery algorithm uses these variables to generate the SQL query.

Next we offer relevant features of these algorithms. First of all, we have separated the graphical part from the query generation part. As a consequence, we can improve the query interface and only the algorithm graphic2CoQueL will have to be modified. The algorithms splitCoQueLQuery and writeTargetQuery will remain invariable.

The algorithm splitCoQueLQuery extracts the entity types, relationship types and attributes involved in the query. In this extraction, the path structure is forgotten, as the translation can be done for the conceptual elements one by one taking into account the er2SQL table.

In some algorithms, there are variables prefixed by l or ll . The prefix l means a list while the prefix ll means a list of lists. For example, l entityTypes is a list of entity types while ll attributes is a list of lists of attributes, one list for each entity type. We add the letter p when dealing with a parameter. Thus, pll means a parameter which is a list of lists.

In the algorithm writeTargetQuery for the language SQL, the clause FROM is obtained from the list of entity types and of the list of relationship types. It is a comma separated list whose items are the tables, encountered in the column table of table er2SQL, corresponding to the entity types and relationship types of the query. If this column is null, nothing is added to the comma separated list. The clause WHERE is obtained by concatenating by AND two kinds of conditions: (1) conditions specified on the entity types and (2) conditions arising from the relationship types. With respect to the first kind, they are obtained by replacing, in the list of conditions, each attribute by its corresponding column as stored in table er2SQL. With respect to the second condition, they are retrieved from the column expression of table er2SQL. If this column is null, nothing is added to the ANDed conditions.

5.3 Examples

Let us suppose the user has specified queries 2 and 3 as in Figures 4 and 5 respectively. Then, the result of applying the algorithm graphic2targetQueryLanguage are the SQL queries shown next.
Query 1. Find the employees who work in the purchasing department in Zaragoza

SELECT *

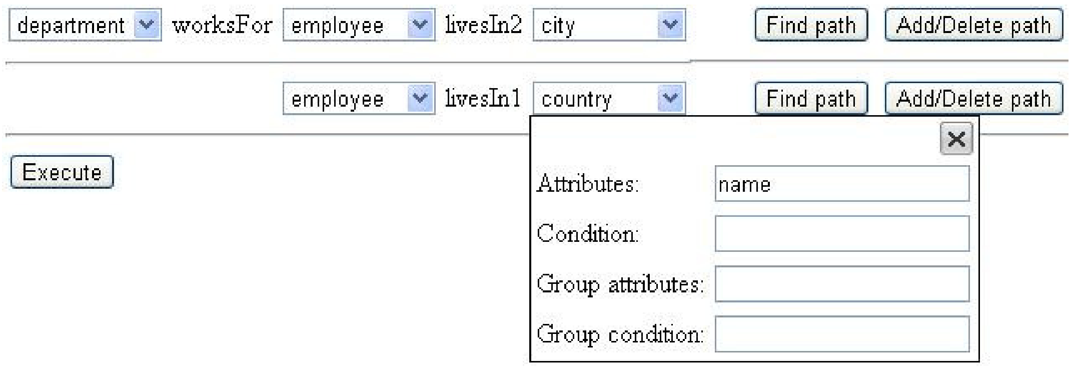
FROM employee, department, situatedIn, city

WHERE employee.idDepartment=department.idDepartment AND

department.idDepartment=situatedIn.idDepartment AND

situatedIn.idCity=city.idCity AND city.name='Zaragoza' AND department.name='purchasing'

Query 2. For each department, find its name, the name of its employees, the name of the city and the country where the employees live



department worksFor employee livesIn2 city

employee livesIn1 country

Attributes: name

Condition:

Group attributes:

Group condition:

Fig. 4.Query 2, about departments, in the CoQueL interface

SELECT department.name, employee.name, country.name, city.name

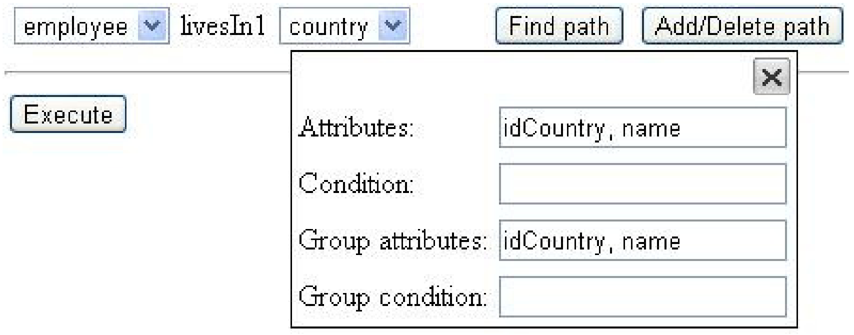
FROM department, employee, country, city

WHERE country.idCountry=employee.idCountry AND

employee.idDepartment=department.idDepartment AND

country.idCountry=employee.idCountry AND city.idCity=employee.idCity

Query 3. Find the name and identifier of the countries where more than five employees live



Attributes: idCountry, name

Condition:

Group attributes: idCountry, name

Group condition:

Fig. 5.Query 3, about countries, in the CoQueL interface

SELECT country.idCountry, country.name

FROM country, employee

WHERE country.idCountry=employee.idCountry

GROUP BY country.idCountry

HAVING COUNT(employee.idEmployee)>5

6. GRAPHICAL INTERFACE

For designing the interface, we have gathered recommendations available in the literature such as the principles of data manipulation [3] or the idea of query abbreviation [7] and we have integrated them into our interface.

The idea of query abbreviation consists of using built-in metaknowledge to determine the paths between the entity types involved in the query so that users do not need to know the conceptual schema. With respect to data manipulation, we have incorporated the principle of offering the user physical actions or labeled button presses instead of complex syntax. With this purpose, we have chosen a form-based interface where the user makes physical actions for specifying the origin and destination of the paths and presses buttons for actions like finding the complete paths involved in the query. The initial aspect of the interface can be seen in Figure 6.

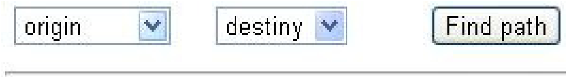


Fig. 6. Initial aspect of the CoQueL interface

At the beginning, the user has to specify the first path of the query. For doing this, (s)he has two options: (1) to choose only the origin entity type or (2) to choose both the origin entity type and the destination entity type. For option (1), when the user clics the ‘Find path’ button, the maximal basic spath whose origin is the selected entity type appears. There, the user selects the appropriate arms of the spath for the query. For option (2), when the user clics the ‘Find path’ button, the collection of lpaths between the entity types origin and destination appears and the user picks one of them for the query. Regardless of the chosen option, at this moment each line of the interface corresponds to a basic lpath.

To complete the rest of the paths involved in the query, the following typical actions are available under the 'Add/Delete path' button for each path: Add path. Add a new path just below the path where the 'Add/Delete path' button is. Its aspect is the same as the first path of the query (Figure 4) and the interaction is as previously described for this first path. Delete path. Delete the path situated next to the 'Add/Delete path' button.

Once every path needed for the query has been chosen, the user must complete the query in the entity types of the paths. To do so, the user double-clicks on the name of the entity types and, in the frame which appears, (s)he adds the attributes, the conditions about attributes, the groups and the conditions about groups. For example, for the query examples number 2 and 3, the specification on entity type country can be seen in Figure 5 and Figure 6.

We are currently implementing a prototype of our CoQueL query system in a laptop with Windows 7, using the Visual C# programming language.

7. RELATED WORK

Query languages for end users have been widely discussed in papers. The first work on this subject was QBE [2]. The paper [3] presents a spreadsheet algebra, adapted from relational algebra, and a spreadsheet interface. The expressive power of expressions in the spreadsheet algebra is the same as that of core SQL single-blocks query expressions. Unlike our paper, both papers lack a conceptual level, which facilitates query writing.

The paper ConQuer [11] inspired our work but unlike ConQuer, we have chosen the E/R model because it is widely extended. ConQuer is a conceptual query language built on ORM models. It enables end users to formulate queries without needing to know how the information is stored in the underlying database. ConQuer queries may be represented as outline queries, schema trees or text.

The papers SQBE [4] and NeuronBank [5] concentrate on a particular data format, unlike our work, which is intended for formats like relational or XML. SQBE [4] is a visual query interface intended for the semantic web, where the data model is RDF and the query languagge is SPARQL. Those users with partial or no knowledge of RDF specify queries visually as a query graph. Then, the algorithm TRANSLATE translates the query graph into a SPARQL query. In NeuronBank [5] a visual web query system is presented aimed at meeting the challenges of extracting information of complex and quickly evolving life science in data. It offers a form-based interface with which queries on an ontology about neurons of different species are specified in the web client.

The paper [7] proposes a conceptual query language, called CQL, built on E/R models where query formulation does not require the user to specify query paths. From the specification, the system derives the corresponding semantically correct full query. Once specified, conceptual queries are translated into SQL. A user-centered approach was adopted in the development of CQL, specifically it was guided through trials and feedback from end-users. Its interface is a bit cluttered, so we have tried to improve it.

The paper [6] deals with query visualization, that is, the process of visualizing queries starting from their SQL expression. Queries are visualized by means of familiar UML notations and incorporate visual metaphors from diagrammatic reasoning. This notation could also be used for specifying conceptual queries. It has been implemented in the QueryViz tool, which is available on the web.

8. CONCLUSIONS AND FUTURE WORK

As more and more collections of data are available on the Internet, end users but not experts in Computer Science demand easy solutions for retrieving data from these collections. In this paper, we have presented a new architecture for querying databases which integrates (1) the CoQueL language, which allows us to specify conceptual queries on an E/R model (2) a graphical interface built on the CoQueL language and (3) a translation from the graphical query to a target language such as XQuery or SQL.

As future work, we plan to extend the expressive power of the CoQueL queries. To date, they are equivalent to SQL single-block query expressions as defined in [3] and we intend CoQueL queries to have the same expressive power as SQL. Second, according to [12], the conceptual query languages have not become widely accepted and one of the reasons is that they lack formal semantics. So, another future work is to provide a formal semantics for the CoQueL language.

REFERENCES

[1] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In SIGMOD Conference, pages 13–24, 2007.

[2] Mosh´e M. Zloof. Query by example. In AFIPS National Computer Conference, volume 44 of AFIPS Conference Proceedings, pages 431–438. AFIPS Press, 1975.

[3] Bin Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng, editors, ICDE, pages 417–428. IEEE, 2009.

[4] Inchul Song and Myoung Ho Kim. Semantic query-by-example for rdf data. In Proc. of Emerging Database Technology, 2009.

[5] Weiling Lee, Rajshekhar Sunderraman, and Paul Katz. A visual web query system for neuronbank ontology. In VISSW 2011, IUI '11, pages –, New York, NY, USA, 2011. ACM.

[6] Wolfgang Gatterbauer. Databases will visualize queries too. PVLDB, 4(12):1498– 1501, 2011.

[7] Vesper Owei. Development of a conceptual query language: Adopting the usercentered methodology. Comput. J., 46(6):602–624, 2003.

[8] Peter P. Chen. The entity-relationship model - toward a unified view of data. ACM Trans. Database Syst., 1(1):9–36, 1976.

[9] E. F. Codd. A relational model of data for large shared data banks. Commun. ACM, 13(6):377–387, 1970.

[10] Ramez Elmasri and Shamkant Navathe. Fundamentals of Database Systems. Addison-Wesley Publishing Company, USA, 6th edition, 2010.

[11] Anthony C. Bloesch and Terry A. Halpin. Conquer: A conceptual query language. In Bernhard Thalheim, editor, ER, volume 1157 of Lecture Notes in Computer Science, pages 121–133. Springer, 1996.

[12] Michael Lawley and Rodney W. Topor. A query language for eer schemas. In Australasian Database Conference, pages 292–304, 1994.

APPENDIX. COMPLETE SYNTAX OF A COQUEL QUERY

entityType = entityTypeName(attributes; conditional-expression; attributes; conditional-expression) attributes = * | attribute-expression

attributeexpression = attribute-list | attribute-name

attribute-list = attribute-list, attribute-name

lpath = [entityType link]* {entityType|spath}

spath = entityType ([&(link{lpath|spath})]+)

CoQueLquery = lpath | spath

conditionalexpression = conditional-term | conditional-expression OR conditional-term

conditional-term = conditional-factor | conditional-term AND conditional-factor

conditional-factor = simple-condition | conditional-expression

simple-condition = comparison-condition | like-condition

like-condition = string-expression LIKE pattern

comparisoncondition = scalar-expression comparison-operator scalar-expression

comparisonoperator = < > <= >= <> =

scalar-expression = numeric-expression | string-expression

numeric-expression = numeric-term | numeric-expression {+|-}numeric-term

numeric-term = numeric-factor|numeric-term{*| /} numeric-factor

numeric-factor = [+|-]primary-number

primary-number = attribute name possibly qualified or numeric literal or aggregate function or numeric expression between parenthesis

string-expression = concatenation | primary-string

concatenation = string-expression || primary-string

primary-string = attribute name possibly qualified or string literal or string expression between parenthesis

entityTypeName = name of some of the entity types of the E/R model

link = link of some of the relationship types of the E/R model

The entry point is CoQueLquery. The symbols [ ] { } |are part of the metasyntax and are never written. [. . .] means one ocurrence at most of the content of the brackets. [. . .]* means zero or more ocurrences of the content of the brackets. [. . .]+ means two or more ocurrences of the content of the brackets. {a|b} means exactly one ocurrence of the elements separated by the vertical bars. The symbols ; & ( ) stand for themselves. The same applies to the operators AND OR ||LIKE =<><=>=<> + − */.