Conceptual Queries using ConQuer–II

A. C. Bloesch and T. A. Halpin

This paper was presented at ER'97: 16th International Conference on Conceptual Modeling.

Formulating non-trivial queries in relational languages such as SQL and QBE can prove daunting to end users. ConQuer is a conceptual query language that allows users to formulate queries naturally in terms of elementary relationships, operators such as "and", "or", "not" and "maybe", contextual for-clauses and object-correlation, thus avoiding the need to deal explicitly with implementation details such as relational tables, null values, outer joins, group-by clauses and correlated subqueries. While most conceptual query languages are based on the Entity-Relationship approach, ConQuer is based on Object-Role Modeling (ORM), which exposes semantic domains as conceptual object types, allowing queries to be formulated via paths through the information space. As a result of experience with the first implementation of ConQuer, the language has been substantially revised and extended to become ConQuer–II, and a new tool, ActiveQuery, has been developed with an improved interface. ConQuer-II's new features such as arbitrary correlation and subtyping enable it to be used for a wide range of advanced conceptual queries.

Introduction and Related Work

A conceptual schema expresses the structure of an application model using concepts familiar to end users, thus facilitating communication between modeler and subject matter experts during the modeling process. Once declared, a conceptual schema can be mapped in an automatic way to a variety of DBMS structures. Although CASE tools are often used for conceptual modeling and mapping, they are rarely used for querying the conceptual model directly. Instead, queries are typically formulated either at the external level using forms, or at the logical level using a language such as SQL, QBE or OQL based on the generic data model (e.g. relational or object-oriented) supported by the DBMS.

Form-based interfaces are limited to simple queries, which can rapidly become obsolete as the external interface evolves. For relational databases, SQL and QBE are more expressive; but non-trivial queries and even queries that are trivial to express in natural language (e.g. who does not speak more than one language?) can be difficult for non-technical users to express in these languages. Moreover, an SQL or QBE query often needs to be changed if the relevant part of the conceptual schema or internal schema is changed, even if the meaning of the query is unaltered. Finally, relational query optimizers ignore many semantic optimization opportunities arising from knowledge of conceptual constraints.

Logical query languages for post-relational DBMS's (e.g. object-oriented and object-relational) suffer the same problems but to a greater extent. For example, their additional structures (e.g. bags and lists) lead to greater complexity in both user formulation and system optimization. Although some proponents of object-oriented query languages such as OQL [4] describe them as conceptual, this is a mistaken viewpoint (e.g. consider their complex fact structures and dereferencing mechanisms). Languages for pre-relational systems are even lower-level, and hence are totally unsuitable for end users.

For such reasons, many *conceptual query languages* have been proposed to allow users to formulate queries directly on the conceptual schema itself. Several of these were surveyed in our earlier paper [2] and others are mentioned in [21], for example Super [1;22], Hybris[24], ERQL[17] and CBQL[16;20;25]. By and large, current conceptual query language tools based on ER or deductive models are challenging for naïve users, and their use of attributes exposes their queries to instability, since attributes may evolve into entities or relationships as the application model evolves.

This instability is avoided by using a query language based on Object-Role Modeling (ORM), a conceptual modeling approach that pictures the application world in terms of objects that play roles (individually or in relationships), thus avoiding the notion of attribute. ORM has a number of closely related versions (e.g. NIAM [26], FORM [9], NORM [7] and PSM [14]) and is similar to the Object-Relationship Modeling (also given the acronym "ORM") approach used in OSM (Object-oriented Systems Modeling) [8]. ORM facilitates detailed information modeling since it is linguistically based, is semantically rich and its notations are easily populated. An overview of ORM may be found in [10; 13], a detailed treatment in [9] and formal discussions in [11; 12].

The use of ORM for conceptual and relational database design is becoming more popular, partly because of the spread of ORM-based modeling tools, such as InfoModeler from InfoModelers Inc. However, as with ER, the use of ORM for conceptual queries is still in its infancy. The first significant ORM-based query language was RIDL [19], a hybrid language with both declarative and procedural aspects. Although RIDL is very powerful, its advanced features are not easy to master, and while the modeling component was implemented in the RIDL* tool, the query component was not supported. Another ORM query language is LISA-D [14], which is based on PSM and has recently been extended to Elisa-D [23] to include temporal and evolutionary support. LISA-D is very expressive but it is technically challenging for end users. Recently an approach based on query by navigation with a stratified hypermedia architecture was proposed to formulate ORM queries as LISA-D path expressions[15], but to date no LISA-D tool has actually been implemented. The basic concept of query by navigation is also exploited in our ActiveQuery tool, but its design is substantially different in other respects.

Like ORM, the OSM approach avoids the use of attributes as a base construct. A prototype has been developed for graphical query language OSM-QL [8] based on this approach. For any given query, the user selects the relevant part of the conceptual schema, and then annotates the resulting subschema diagram with the relevant restrictions to formulate the query. Negation is handled by adding a frequency constraint of "0", and disjunction is introduced by means of a subtype-union operator. Projection is accomplished by clicking on the relevant object nodes and then on a mark-for-output button. One of the strengths of OSM-QL is its support for quantified conditions. For example, to restrict employees to those that speak at least 3 and at most 5 languages, one simply adds the frequency constraint “3:5” to the Employee end of the relationship Employee speaks Language.

While our version of ORM includes similar frequency constraints in the modeling notation (e.g. “3-5”), ConQuer–II instead uses a count function which can be used to construct a frequency constraint. Once such a constraint has been defined in ConQuer–II, it can be called as a macro in later queries.

From our research of conceptual query tools, we believe OSM-QL to be closest in power and ease of use to our own. Based on the published details [8] of OSM-QL, it appears that ConQuer–II is more expressive and also easier for novice users. For example, it is unclear whether OSM-QL can support arbitrary projections (e.g. of functions such as count) or arbitrary correlations, and there does not appear to be the equivalent of our “maybe” operator (corresponding to conceptual left outer joins). We use “not” and “or” instead of 0 frequencies and subtype unions, and do not require the user to have any familiarity with the conceptual schema or our diagram notation. Another fundamental difference between the OSM-QL tool and our tool is that we map any ConQuer–II query into a sequence of one or more SQL queries for execution on the user’s commercial DBMS (Informix, Oracle, MS Access, etc.). In contrast, the OSM-QL prototype implements the query only in terms of its own database language.

The name “ConQuer” derives from “CONceptual QUERy”. The first version of ConQuer was commercially released in InfoAssistant, a Windows-based tool that has received positive industry reviews. Based on our own experience and user feedback, we redesigned the language and user interface for greater expressibility and usability. The new tool is called ActiveQuery, and should be available by late-1997 from InfoModelers as an OLE control for Windows applications. As well as complying with Microsoft’s user interface standards, the tool provides an intuitive interface for constructing almost any query that might arise in an industrial database setting. Typical queries can be constructed by just clicking on objects with the mouse, and adding conditions.

The rest of this paper provides an overview of ConQuer–II as supported by ActiveQuery. The next section explains how the language is based on ORM, and illustrates how queries are formulated and mapped to SQL. The following three sections describe ConQuer–II’s improvements to ConQuer, the query engine, and formal semantics, and the conclusion summarizes the main contributions and outlines future research.

Formulating and Mapping Conceptual Queries

A discussion of the first version of ConQuer (ConQuer–I) and its associated tool may be found in [2]. At that stage, queries were allowed only on schemas that had been reverse engineered (see Figure 1). While the reverse engineering was automatic, the default names generated for schema components were based on the relational schema and were thus often unintuitive. Further, the domain information was based on foreign keys and thus was often incomplete. A separate tool (FactBuilder) allowed domain and predicate text to be improved, and domains to be merged.

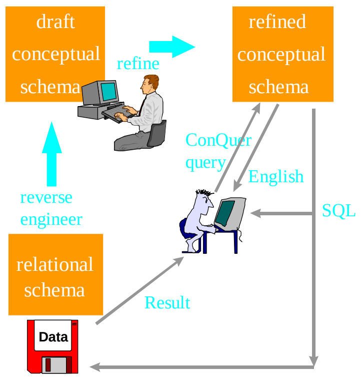

Figure 1: Existing databases may be reverse-engineered then queried.

The ConQuer–II query tool is completely new. Queries are now based on InfoModeler's meta-data, allowing InfoModeler based conceptual models to be directly queried. ConQuer–II queries are automatically translated into English (allowing users to validate the query) and automatically translated into back-end specific, SQL chain queries (for curious users, and to retrieve query results from the designated database).

The underlying ORM framework

We now briefly illustrate and motivate the ORM modeling framework on which ConQuer–II queries are based. Figure 2 is a simple ORM schema fragment. Object types are shown as named ellipses. Entity types have solid ellipses with their simple reference schemes abbreviated in parenthesis (these references are often unabbreviated in queries). For example, “Academic (empnr)” abbreviates “Academic is identified by EmpNr”.

If an entity type has a compound reference scheme, this is shown explicitly using an external uniqueness constraint (a circled “P” denotes primary reference). For example, a degree is identified by combining its degreecode (e.g. “PhD”) with the university that awarded it (e.g. UCLA). Value types have dotted ellipses (e.g. “Degreecode”), and a “+” indicates numeric reference.

Predicates are shown as named role sequences, where each role is depicted as a box. A role is just a part in a relationship. In the example, all the relationships are binary (two roles) except for the ternary (three role): Academic was awarded Degree in Year. Predicates may have any arity (number of roles) and may be written in mixfix form. A relationship type not used for primary reference is a fact type. An *n*-ary relationship type has *n*! readings, but only *n* are needed to guarantee access from any object type. Figure 2 shows forward and inverse readings (separated by “ / ”) for some of the relationships.

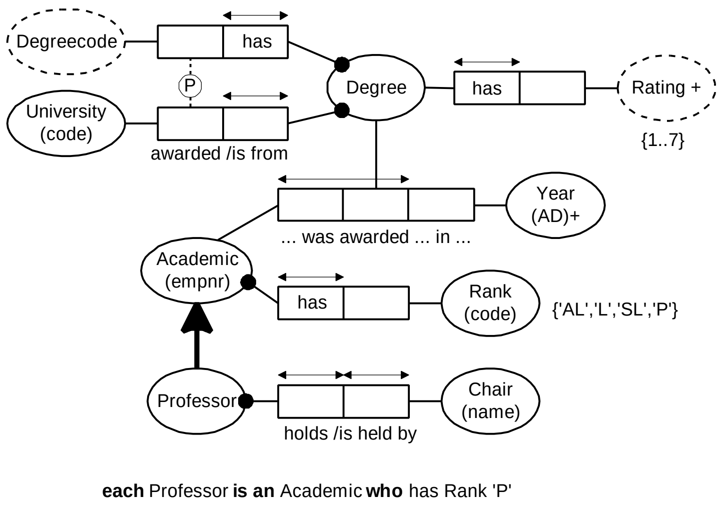

Figure 2: An ORM conceptual schema.

An arrow-tipped bar across a role sequence depicts an internal uniqueness constraint. For example, each academic has at most one rank, but the same rank may apply to more than one academic, and an academic may be awarded a given degree in at most one year. A black dot connecting a role to an object type indicates that the role is mandatory (i.e. each object in the database population of that object type must play that role). The possible values for a value type may be listed in braces beside the relevant type (e.g. Rank and Rating). Subtypes are connected to their supertype(s) by arrows, and given formal definitions (e.g. Professor). Subtype definitions themselves form one class of named ConQuer–II queries. ORM has many other kinds of constraint [9], but these are ignored in this paper since they are not germane to our discussion.

No use is made of attributes. This helps with natural verbalization, simplifies the framework, avoids arbitrary or temporary decisions about whether some feature should be modeled as an attribute, and lengthens the lifetime of conceptual queries since they are not impacted when a feature is remodeled as a relationship or attribute. This semantic stability of ORM models, and hence ORM queries, gives it a major advantage over ER, OO and lower level approaches. For examples illustrating this claim, see [2].

Since ORM conceptual object types are semantic domains, they act as semantic “glue” to connect the schema. This facilitates not only strong typing but also query navigation through the information space, enabling joins to be visualized in the most natural way. When desired, attributes (e.g. degree_awarded) can be introduced as derived concepts, based on roles in the underlying ORM schema. Various abstraction mechanisms can also be applied to support queries at a higher level [3].

Sample Queries and SQL Mapping

Although ConQuer–II queries are based on ORM, it is not necessary for the user to be familiar with ORM or its notation. The query is shown in textual (outline) form (basically as a tree of predicates connecting objects) with the underlying constraints hidden, since they have no impact on the meaning of the query. Moreover, the user can construct a query without any prior knowledge of the schema.

On opening a model for browsing, the user is presented with an object pick list. When an object type is dragged to the query pane, another pane displays the roles played by that object in the model. The user drags over those relationships of interest. Clicking an object type within one of these relationships causes its roles to be displayed, and the user may drag over those of interest, and so on. In this way, users may quickly declare a query path through the information space, without any prior knowledge of the underlying data structures. Users may also initially drag across several object types. The structure of the underlying model is then used to automatically infer a reasonable path through the information space.

Items to be displayed are indicated with a tick "✓": these ticks may be toggled on/off as desired. The query path may be restricted in various ways by use of operators and conditions. As a simple example, consider the query: *For those academics who were awarded a degree that has a rating above 5, list the academics and those degrees.* This may be set out as the following ConQuer outline:

```
Q1    ✓Academic
          +-  was awarded ✓Degree in Year
                               +-  has Rating > 5
```

In terms of the ORM schema in Figure 2, this query is a path from Academic through Degree to Rating, extended by a path through the system predicate ">" to the object 5, and then a projection is made on Academic and Degree, which are dereferenced to their identifiers for display of the result. Moving through an object type (e.g. Degree) corresponds to a conceptual join. Since Degree has a composite identification scheme this maps to a composite join in SQL.

Degree (degreecode, university, rating)

Awarded (empnr, degreecode, university, awardyear)

6

Academic (empnr, rank, [chair]^1)

^{1} exists iff rank = 'P'

Figure 3: The relational schema mapped from the ORM conceptual schema I in Figure 2.

Using the Rmap algorithm [9; 18], our conceptual schema maps to the relational schema shown in Figure 3 for simplicity, domains and value constraints are omitted). Keys and uniqueness constraints are indicated by underlining. Optional columns are shown in square brackets. A subset constraint (e.g. foreign key constraint) is shown as a dotted arrow. By default, Rmap absorbs functional relationships of subtypes into their supertype table. Here the numbered qualification on chair enforces the subtype definition and the relevant mandatory role constraint.

The SQL generated for query Q1 is shown below. Notice how the conceptual query shields the user from details about Degree's composite reference scheme. Here the relational join is a result of the same degree playing two roles which map to separate tables, with no foreign key connection. In contrast to our semantic domain approach, some query tools require foreign keys to perform a join, and even force the user to specify what kind of join (e.g. inner or outer) is associated with a foreign key connection: the limitations of such an approach are obvious.

```
S1     select X1.empnr, X1.degreecode, X1.university
       from Awarded X1, Degree X2
       where X1.degreecode = X2.degreecode and
             X1.university = X2.university and
             X2.rating > 5
```

Basic queries, mappings and semantic optimization for ConQuer were discussed in [2]. We now illustrate some of the new features introduced in ConQuer–II. Connections from a subtype to a supertype appear as an "is" predicate, which may be navigated like any other predicate. For example, the following query asks for those academics who were not awarded any degree from the University of Queensland (UQ) but are professors who hold the informatics chair. The verbalization generated for the query provides a more natural reading. For example, "and" is inserted for conjunctions, the "not" operator is moved after "was", and pronouns and articles are added.

```
Q2     ✓Academic
             +- is Professor
             ¦           +- holds Chair = Informatics
             +- not was awarded Degree in Year
                                     +- is from University = UQ
```

Based on the default relational schema in Figure 1, the following SQL code is generated:

```
S2     select X1.empnr
       from Academic X1
       where X1.chair = 'Informatics' and
             X1.empnr not in (select X2.empnr from Awarded X2
                              where X2.university = 'UQ')
```

If the mapping of the actual model is modified to store the chair fact type in a separate table for professor, the conceptual query remains unaltered although the SQL generated is now different (in the first from-clause of S2, Academic is replaced by Professor)– another example of semantic stability.

ConQuer used a "maybe" operator to perform conceptual left outer joins but ignored any conditions after the operator. ConQuer–II extends the semantics of "maybe" to evaluate such conditions. For example, the following query lists each academic as well as their degrees (if any) that are worth more than a 5 rating.

```
Q3     ✓Academic
             +- maybe was awarded ✓Degree in Year
                                        +- has Rating > 5
```

In SQL-92 this can be translated as a single SQL query as follows:

```
S3    select X1.empnr, X2.degreecode, X2.university
      from Academic X1 left outer join (
                select * from Awarded X2, Degree X3
                where X2.degreecode = X3.degreecode and
                     X2.university = X3.university and
                     X3.rating > 5)
           on X1.empnr = X2.empnr
```

Unfortunately, target DBMS's are rarely orthogonal in their support for joins, so in practice this query is translated as a sequence of SQL queries, where the inner join is first computed and the intermediate result is used as the right-hand argument of the outer join in the final query.

The next two examples are based on a different schema concerning employees. ConQuer included basic support for functions and grouped queries, but had complex rules for disambiguation, and required conditions to be added in separate windows. ConQuer–II now has much greater support for such queries, provides for-clauses to disambiguate the grouping criteria, and enables full queries, including all conditions, to be entered on the one query pane. For example, query Q4 lists each department and those of its employees whose maximum individual rating exceeds the average of his/her department (each employee may have many ratings). Notice how the different grouping criteria are captured in the for-clauses. We leave it as an exercise for the interested reader to provide the SQL.

```
Q4    ✓Department
          +-  employs ✓Employee
                      +-  achieves Rating
                                   +-  max(Rating) for Employee >
                                         avg(Rating) for Department
```

The most significant enhancement to ConQuer–II is its ability to deal with *correlations* of arbitrary complexity. For example, consider the query: *Who owns a car, and does not drive more than one of those cars (that he/she owns)?*. In English, correlation is often achieved through pronouns. Here there is a correlation on cars ("those") as well as employees ("he/she"). This query may be formulated as Q5. When an object type is referenced more than once in the query body, the system automatically provides subscripts that the user can modify. Object variables with identical subscripts are correlated. This is used here to correlate cars (Car_1). The for-clause has only one instance of Employee in the query body to reference, so no subscripts are needed to perform the correlation for employees.

```
Q5    ✓Employee
          +-  owns Car1
          +-  not drives Car1
                        +-  count(Car1) for Employee > 1
```

Equivalent SQL is shown below. Because the correlation stems from a function argument inside a negated function subquery, the correlation concerns membership in a set, not just equality with an outer instance (see italicized code).

```
S5    select X1.empnr
      from Owns X1
      where X1.empnr not in (
            select X2.empnr
            from Drives X2
            where X2.car in (select X3.car from Owns X3
                                   where X3.empnr = X1.empnr)
            group by X2.empnr
            having count(X2.car) > 1)
```

ConQuer–II is capable of far more complex queries than cited here. In this paper we are more concerned with a clear exposition of our basic approach and its rationale rather than with providing a complete coverage of the language. The next two sections provide an overview of the query engine and formal semantics.

Improvements in Conquer–II

The design of ConQuer–II focused not only on the weaknesses of ConQuer but also the weakness in the design process that caused them. The most important design decision was to keep ConQuer–II semantics as simple and as standard as possible. To reduce the chance of error and speed development of the SQL generator, it was decided to use compiler techniques as much as possible since they embody a wealth of experience in a similar domain.

The new design sought a semantically clear and well founded core language. Sorted finitary first-order logic with schema comprehension proved ideal. Basing the SQL generation on an unambiguous orthogonal core language greatly simplified the generator's design. ConQuer–II's query representations like outline queries are supported by translating them into the core language before further processing. Once again, the clear semantics of ConQuer–II greatly simplified the task.

Because the query language is based around a well understood logic, there is a well developed literature to draw upon. For example, languages like ConQuer–II have a well developed theory of the power of various constructs, making it clear which constructs are needed to improve expressive power. Further, semantic optimization algorithms are readily available for ConQuer–II like languages.

ConQuer–II's outline queries reflect the constructs available in the core language. For users of outline queries, the major ConQuer–II extensions are support for arbitrary variable correlations, quantifiers, arbitrary grouping criteria, reuse of queries (macros) and the removal of many artificial restrictions on queries.

In ORM terms, ConQuer–II queries define derived predicates. Thus existing queries may be reused as if they were predicates in the conceptual model. The user interface of ActiveQuery does not distinguish between these derived predicates and the conceptual model's predicates. This presents users with a uniform interface where they may reuse queries to form more powerful queries.

Our experience with constructing queries using InfoAssistant highlighted several weaknesses with the old user interface. Consequently, ActiveQuery's user interface was redesigned. All query information is now displayed with the query rather than hidden in property sheets, arbitrary expressions may now be used in queries, and support has been added for handling large schemas (e.g. conceptual paths may be inferred from their endpoints).

The ConQuer–II Query Engine

ConQuer–II is based on the domain relational calculus or, equivalently, sorted finitary first-order logic with schema comprehension. ConQuer–II extends the logic with various constructs to make queries easier to express. For example, ConQuer–II has a modal-like operator *maybe* that corresponds to a conceptual left outer join, and also has summary functions like maximum.

These extensions undermine the first-order basis of ConQuer–II and make it referentially opaque (i.e. the value of an expression cannot be derived solely from the value of its immediate parts). However, there is a straightforward translation of ConQuer–II queries into a first-order language. Thus, ultimately, these extensions are just a matter of convenience rather than a fundamental change in the power or basis of the language. Internally ConQuer–II queries are expanded to the base logic before any processing is done.

ConQuer–II queries consist of a sequence of named set comprehensions and operations (*subqueries*). Each subquery may refer to earlier subqueries with the final subquery returning the result set of the overall query.

Tools that use the ConQuer–II query engine translate between their external representation and ConQuer–II. For example, ActiveQuery translates between an English-like linearized version of ConQuer–II and ConQuer–II. QBE-like, English text, form based, and direct interfaces are all planned as future interfaces to the ConQuer–II query engine.

There is little research experience in implementing query engines like ConQuer–II's. However, the implementation of translators for programming languages is well understood. Since the task of translating ConQuer–II queries into SQL is similar to programming language translation, it was felt prudent to base the ConQuer–II query engine on programming language translators.

Internally, ConQuer–II queries are represented as abstract syntax trees. Attributes may be associated with nodes, and mechanisms exist for computing synthesized and inherited attributes as well as transforming nodes. Most of the SQL translation is done through visitor classes which calculate attributes and perform tree transformations. In order to simplify SQL translation, ConQuer–II queries are first translated into an abstract SQL dialect and then transformed into the backend-specific SQL. Currently, Informix (Online, SE and Universal), Oracle, DB2, MS SQL Server, Sybase SQL Server, MS Access, FoxPro, Paradox, d-Base, SQL Anywhere and generic ODBC databases are supported. This approach mimics the intermediate languages often used in multi-platform compilers.

The ConQuer–II query engine is written in C++ and consists of about 25 000 lines of code, 15 000 lines of comments, 12 000 blank lines and 210 classes. Further, the ConQuer–II query engine shares some 83 000 lines of code with InfoModeler. ActiveQuery adds some 35 000 lines of, essentially GUI, code based on Microsoft's MFC library.

A Semantics for ConQuer–II

ConQuer–II is a sugared version of sorted finitary first-order logic with set comprehension (σ). In this section, a semantics, based on bag comprehension, is given for ConQuer–II's extensions to σ. The treatment is necessarily brief, but it should be clear how the semantics could be completely formalized. Note that conceptual nulls can never occur, so there is no need to use Lukasiewicz's (or any other) three valued logic.

A ConQuer–II query may be interpreted as specifying the contents of a set, via set comprehension. For example, the query: *Which cars are red?* would correspond to the expression:

{x : Car | ∃ y : Color; z : ColorName (x has y ∧ y has colorname z ∧ z = 'red')}

In general, a ConQuer–II query corresponds to the straightforward translation of the query into σ with the ticked object types quantified over by the set comprehension operator and the non-ticked object types by an existential quantifier (unless already bound by a quantifier).

Care must be taken in interpreting the "maybe" operator. Expressions of the form "maybe α", where α is some path expression, should be translated as:

$\alpha' \vee (\neg\, \alpha \wedge x1 = \square \wedge x2 = \square \wedge \ldots \wedge xn = \square)$

where α' is the translation of α; x_1, x_2, …, x_n are the selected object types in α; and $\square$ is a blank in the result set (relationally a null).

A semantics for the summary (bag or aggregate) functions of ConQuer–II can be given as follows. For a given aggregate expression $\iota(\alpha_1, \ldots, \alpha_n; \gamma_1, \ldots, \gamma_m)$ (where $\alpha_1, \ldots, \alpha_n$ are arguments and $\gamma_1, \ldots, \gamma_m$ are grouping criteria) construct a labeled bag v corresponding to the ConQuer–II query where all expressions involving summary functions have been elided and the bag's tuples consist of the object types in the function's grouping criteria $(\gamma_1, \ldots, \gamma_m)$ and the arguments to the function $(\alpha_1, \ldots, \alpha_n)$.

For example, consider the query: *What are the branches and total salary costs of branches with a total salary cost of more than $1 000 000?*, which may be expressed in outline form as:

```
✓ Branch
     +-  employs Employee
                    +-  earns ✓ total(Salary) for Branch > 1 000 000
```

The corresponding labeled bag is:

let S = [′x : Branch, y : $value • z : Employee, w : Salary |
 x employs z ∧ z earns w ∧ w has $value y≤].

Where the notation $[\prime x_1{:}\tau_1, \ldots, x_n{:}\tau_n \bullet\ x_{n+1}{:}\tau_{n+1}, \ldots, x_m{:}\tau_m \mid \rho \leq]$ means the bag formed by discarding $x_{n+1}, \ldots, x_m$ from the tuples in the set $\{x_1{:}\tau_1, \ldots, x_m{:}\tau_m \mid \rho\}$ (i.e. projecting on $x_1, \ldots, x_n$). The query is thus equivalent to:

$$\{x : \text{Branch}, u : \Re \mid \langle x\rangle \in S1 \wedge u = (\Sigma\langle x', y\rangle \in S\ y \mid x = x') \wedge u > 1\,000\,000\}$$

where $\Re$ is the set of reals; $S_{i,\ j,\ \ldots,\ k}$ is a set of tuples made up of the i'th, j'th, … , k'th entries in each tuple of the bag S; and $(\Sigma_{\langle x,\ y,\ \ldots,\ z\rangle \hat{I}\ S}\ \alpha \mid \rho)$ is, for each $\langle x, y, \ldots, z\rangle$ in of the bag S, the sum of every α such that ρ holds.

In general, for a given grouping criterion each object type in the criterion and the arguments of the corresponding aggregate functions are quantified over by the bag comprehension, a conjunct links selected object types to the corresponding elements of S, a series of conjuncts specifies the value of each aggregate function and a series of conjuncts corresponds to any conditions involving aggregate functions.

Future Plans

This paper has discussed ConQuer–II, an ORM-based conceptual query language that enables end users to formulate queries in a natural way, without knowledge of how the information is stored in the underlying database. The benefits of this approach were highlighted and a semi-formal semantics provided.

The main extension planned for the ConQuer–II query engine is recursive completeness. First-order languages like ConQuer–II lack the expressive power of recursive languages. The most probable extension is some kind of fixed-point operator. Clearly, the ConQuer–II query engine could no longer target just SQL dialects and would need to target stored procedure languages.

The ConQuer–II query engine is designed to support object-relational extensions to RDBMS's. In the future, support for these extensions will become part of ConQuer–II. However, support for collection types represents a significant challenge since some queries, involving columns containing collection types, expressible in ConQuer–II require very complex procedural code to be generated. By contrast, row types, and user defined objects and functions can be easily supported.

ORM has a rich constraint language. The ConQuer–II query engine makes use of these constraints to produce more efficient SQL than would otherwise be possible. Much more powerful semantic optimizations are planned for the future. Because of ConQuer–II's first-order basis and clean semantics it will be possible to make use of the very efficient and powerful techniques that have been developed by the automated theorem proving community.

ConQuer–II seems ideal for expressing *ad hoc* constraints in an ORM model. Extensions to InfoModeler (InfoModelers Inc.'s conceptual modeling tool) are planned that would allow *ad hoc* constraints and derivation rules to be constructed with ActiveQuery. ConQuer–II's backend independence and semantic stability mean that such constraints and derivation rules can be expected to retain their meaning under schema transformations and across a variety of backends.

InfoModeler allows users to mix IDEF1X (an ER dialect) models with ORM models. IDEF1X models are internally represented as ORM models but displayed as IDEF1X models. A future release of InfoModeler will allow users to annotate IDEF1X models with the missing predicate text. Thus ActiveQuery will support both ER, ORM and mixed ER/ORM based conceptual queries. Such queries are already possible, but IDEF1X users must be content with the default predicate text.

References

1. Auddino, A., Amiel, E. & Bhargava, B. 1991 'Experiences with SUPER, a Database Visual Environment', *DEXA '91 Database and Expert System Applications*, pp.172-178.

2. Bloesch. A.C. & Halpin, T.A. 1996, 'ConQuer: a conceptual query language', *Conceptual Modeling – ER'96*, Springer LNCS, no. 1157, pp. 121–33.

3. Campbell, L.J, Halpin, T.A. & Proper, H.A. 1996, 'Conceptual schemas with abstractions: making flat conceptual schemas more comprehensible', *Data and Knowledge Engineering*, vol. 20, Elsevier Science, pp. 39-85.

4. Cattell, R.G.G. (ed.) 1994, *The Object Database Standard: ODMG–93*, Morgan Kaufmann Publishers, San Francisco.

5. Date, C.J. 1996, 'Aggregate functions', *Database Prog. & Design*, vol. 9, no. 4, Miller Freeman, San Mateo CA, pp. 17-19.

6. Date, C.J. & Darwen, H. 1992, *Relational Database: writings 1989-1991*, Addison-Wesley, Reading MA, esp. Chs 17-20.

7. De Troyer, O. & Meersman, R. 1995, 'A logic framework for a semantics of object oriented data modeling', *OOER'95: Object-Oriented and Entity-Relationship Modeling*, Springer LNCS, no. 1021, pp. 238-49.

8. Embley, D.W., Wu, H.A., Pinkston, J.S. & Czejdo, B. 1996, 'OSM-QL: a calculus-based graphical query language', Tech. Report, Dept of Comp. Science, Brigham Young Univ., Utah.

9. Halpin, T.A. 1995, *Conceptual Schema and Relational Database Design, 2nd edn*, Prentice-Hall Australia, Sydney.

10. Halpin, T.A. & Orlowska, M.E. 1992, 'Fact-oriented modelling for data analysis', *Journal of Inform. Systems*, vol. 2, no. 2, pp. 1-23, Blackwell Scientific, Oxford

11. Halpin, T.A. & Proper, H.A. 1995, 'Subtyping and polymorphism in Object-Role Modeling', *Data and Knowledge Engineering*, vol. 15, Elsevier Science, pp. 251-81.

12. Halpin, T.A. & Proper, H. A. 1995, 'Database schema transformation and optimization', *OOER'95: Object-Oriented and Entity-Relationship Modeling*, Springer LNCS, no. 1021, pp. 191-203.

13. Halpin, T.A. 1996, 'Business Rules and Object-Role Modeling', *Database Prog. & Design*, vol. 9, no. 10, Miller Freeman, San Mateo CA, pp. 66-72.

14. Hofstede, A.H.M. ter, Proper, H.A. & Weide, th.P. van der 1993, 'Formal definition of a conceptual language for the description and manipulation of information models', *Information Systems*, vol. 18, no. 7, pp. 489-523.

15. Hofstede, A.H.M. ter, Proper, H.A. & Weide, th.P. van der 1996, 'Query formulation as an information retrieval problem', *The Computer Journal*, vol. 39, no. 4, pp. 255-74.

16. Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M., Eherer, S., 1995, *ConceptBase– a Deductive Object Base for Meta Data Management*, Journal of Intelligent Information Systems, Special Issue on Advances in Deductive Object-Oriented Databases, vol. 4, no. 2, 167-192.

17. Lawley, M. & Topor R. 1994, 'A Query Language for EER Schemas', *ADC'94 Proceedings of the 5th Australian Database Conference*, Global Publications Service, pp. 292-304.

18. McCormack, J.I., Halpin, T.A. & Ritson, P.R. 1993, 'Automated mapping of conceptual schemas to relational schemas', *Advanced Inf. Sys. Eng: CAiSE'93*, Springer LNCS, no. 685, pp. 432-48.

19. Meersman, R. 1982, 'The RIDL conceptual language', Research report, Int. Centre for Information Analysis Services, Control Data Belgium, Brussels.

20. Mylopoulos, J., Borgida, A., Jarke, M. & Koubarakis, M., 1990, *Telos: a language for representing knowledge about information systems*, ACM Transactions Information Systems vol. 8, no 4.

21. Ozsoyoglu, G. & Wang, H. 1993, 'Example-based graphical database query languages', *Computer*, vol. 26, no. 5, pp. 25-38.

22. Parent, C. & Spaccapietra, S. 1989, 'About Complex Entities, Complex Objects and Object-Oriented Data Models', *Information System Concepts– An In-depth Analysis*, Falkenberg, E.D. & Lindgreen, P., Eds., North Holland, pp. 347-360

23. Proper, H.A. & Weide, Th. P. van der 1995, 'Information disclosure in evolving information systems: taking a shot at a moving target', *Data and Knowledge Engineering*, vol. 15, no. 2, pp. 135-68, Elsevier Science.

24. Rosengren, P. 1994, 'Using Visual ER Query Systems in Real World Applications', *CAiSE'94: Advanced Information Systems Engineering*, Springer LNCS, no. 811, pp. 394-405.

25. Staudt, M., Nissen, H.W., Jeusfeld, M.A. 1994, *Query by Class, Rule and Concept*. Applied Intelligence, Special Issue on Knowledge Base Management, vol. 4, no. 2, pp. 133-157

26. Wintraecken, J.J.V.R. 1990, *The NIAM Information Analysis Method: Theory and Practice*, Kluwer, Deventer, The Netherlands.

This paper is made available by Dr. Terry Halpin and is downloadable from www.orm.net.