Conceptual Queries

by Dr. Terry Halpin, BSc, DipEd, BA, MLitStud, PhD
Director of Database Strategy, Visio Corporation

This paper first appeared in vol. 26, no. 2 of Database Newsletter and is reproduced by permission.

Formulating non-trivial queries in relational languages such as SQL or QBE can prove daunting to end users. ConQuer, a new conceptual query language based on Object Role Modeling (ORM), enables users to pose complex queries in a readily understandable way, without needing to know how the information is stored in the underlying database. This article highlights the advantages of conceptual query languages such as ConQuer over traditional query languages for specifying queries and business rules.

Four query levels

There are four main levels at which humans may communicate with an information system:

- External
- Conceptual
- Logical
- Physical

The *external* level deals with the actual interfaces and input/output representations used to work directly with the system (e.g. screen forms and printed reports). At the *conceptual* level, the information is expressed in its most fundamental form, using concepts and language familiar to the users (e.g. Employee drives Car) and ignoring implementation and external presentation aspects. At the *logical* level, a commitment is made to the general type of data model to be used for storage (e.g. relational or object-oriented) and the information is expressed using the logical constructs of that model (e.g. tables and keys). At the *physical* level, a specific DBMS is chosen (e.g. MS Access or DB2) and all the detailed internal details are fleshed out (e.g. indexes and clustering).

Since a conceptual schema expresses the structure of an application from a human rather than a machine perspective, it facilitates communication between modeler and subject matter experts during the modeling process, and it can be mapped automatically to a variety of DBMS structures. Although software 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 or QBE.

Query-By-Form (QBF) enables users to enter queries directly on a screen form, by entering appropriate values or conditions in the form fields. This form-based interface is well suited to simple queries where the scope of the query is visible on a single form, and no complex operations are involved. However this cannot be used to express complicated queries. Moreover, saved QBF queries may rapidly become obsolete as the external interface evolves. For such reasons, QBF is too restrictive for serious work.

For relational databases, SQL and QBE (Query-By-Example) are more expressive. However, complex queries and even queries that are easy 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 constraints.

Logical query languages for post-relational DBMS's (e.g. object-oriented and object-relational) suffer similar problems. Their additional structures (e.g. sets, arrays, bags and lists) often lead to greater complexity in both user formulation and system optimization. For example, OQL [3] extends SQL with various functions for navigation as well as composing and flattening structures, thus forcing the user to deal directly with the way the information is stored internally. At the physical level, programming languages may be used to access the internal structures directly (e.g. using pointers and records), but this very low level approach to query formulation is totally unsuitable for end users.

Conceptual query languages

Given the disadvantages of query formulation at the external, logical or physical level, it is not surprising that many *conceptual query languages* have been proposed to allow users to formulate queries directly on the conceptual schema itself [1, 2]. Most of these language proposals are academic research topics, with at best prototype tool support. One commercial tool, English Wizard, provides some ability for users to enter queries directly in English, but the tool currently suffers from problems with ambiguity and expressibility, as well as the correctness of its SQL generation. By and large, current conceptual query language tools based on Entity-Relationship (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 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 [7] and a detailed treatment in [5].

The use of ORM for conceptual and relational database design is becoming more popular, partly because of the spread of ORM-based modeling tools. 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 [9], 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 [8]; although very expressive, it is technically challenging for end users, and currently lacks tool support.

Like ORM, the OSM (Object-oriented Systems Modeling) approach avoids the use of attributes as a base construct. An academic prototype has been developed for graphical query language OSM-QL [4] 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.

Another recent ORM query language is ConQuer (the name derives from “CONceptual QUERy”). ConQuer is more expressive than OSM-QL [2], easier for novice users, and its commercial tool transforms conceptual queries into SQL queries for a variety of back-end DBMSs. Moreover, the ConQuer tool does not require the user to be familiar with the conceptual schema or the ORM diagram notation. The first version of ConQuer was released in InfoAssistant [1]. Feedback from this release led to the redesign of both the language and the user interface for greater expressibility and usability, resulting in a new tool called ActiveQuery [2], a restricted version of which is available 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 suggests some design principles for conceptual query languages, and then illustrates how these principles are realized in ConQuer, as supported by ActiveQuery. A brief outline of the underlying ORM framework is included, as well as examples of how queries are formulated and mapped to SQL. Finally some examples are given of how the query language can also be used to provide high level declaration of business rules.

Language design criteria

The following four criteria were used in designing the ConQuer language and tool support, and seem appropriate for conceptual query languages in general.

- Semantic strength
- Semantic clarity
- Semantic relevance
- Semantic stability

Semantic strength is a measure of a language’s expressibility (i.e. the range of queries that can be expressed in the language). Ideally, the language should allow you to formulate any question that is relevant to your application. In practice, something less than this ideal is acceptable. For most business applications, if the language can express whatever is possible to formulate as a sequence of SQL-89 queries, this is often good enough. In more complex cases, this might not be adequate. For example, a bill of materials query requires recursion, which while supported by recursive union in the long awaited SQL3 standard, is still not available in many commercial SQL dialects. ActiveQuery was designed to translate ConQuer queries into a sequence of SQL statements on the back end DBMS, and hence is limited in practice by the power of the chosen SQL back end. In comparison with the low expressive power of QBF however, this is a very mild limitation.

Semantic clarity is a measure of how easy it is to understand and use the language. To begin with, the language must be unambiguous (i.e. there is only one possible meaning). Since any ConQuer query corresponds to a qualified path through an ORM schema, where all the object types and predicates are well defined, the meaning of the query is essentially transparent. As we discuss shortly, the ActiveQuery tool automatically reveals the relevant part of the application to the user, so that ConQuer queries can be formulated without requiring any prior knowledge of the information space. Although this context revelation is a feature of the tool rather than the language, even a manual formulation of ConQuer queries requires no sub-conceptual knowledge (e.g. knowledge about how the information is actually stored in a database). This is in sharp contrast to a query language such as SQL, QBE or OQL, where the query needs to be formulated in terms of the storage structures themselves.

Semantic relevance requires that only the information relevant to the intent of the query needs to be stated. In order to formulate a query, the user must not be forced to include other features of the application that have no bearing on the question that he or she wants to ask.

Semantic stability is a measure of how well queries retain their original intent in the face of changes to the application. Because ConQuer queries are based on ORM, they continue to produce the desired result so long as their meaning endures. In other words, you never need to change a ConQuer query if the English meaning of the question still applies. In particular, ConQuer queries are not impacted by typical changes to an application, such as addition of new fact types or changes to constraints or the relative importance of some feature. This ensures *semantic independence* (i.e. the conceptual queries are independent of changes to underlying structures when those changes have no effect on meaning).

If the discussion so far seems pretty abstract or hard to follow, it should all become clear with a few examples. The rest of the article is mainly concerned with illustrating these four design criteria with sample queries based on a small application. The underlying ORM schema for this application is explained in the next section.

A sample ORM schema

Although knowledge of the ORM diagram notation is not needed to formulate ConQuer queries, some familiarity with it will help you to understand the basis of the query technology. A ConQuer query can be applied only to an ORM schema. Using a software tool, an ORM schema may be entered directly, or instead reverse engineered from an existing logical schema (e.g. a relational or object-relational schema). While reverse engineering is automatic, some refinement by a human improves the readability (e.g. the default names generated for predicates are not always as natural as a human can supply).

ORM is a bit like ER without attributes. If you are familiar with ER, just use a relationship instead whenever tempted to portray some feature as an attribute, and you have the basic ORM view of the world as a set of objects playing roles (parts in relationships).

Figure 1 is an ORM conceptual schema fragment for an application about a company with branches in various countries. 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, “Car (regnr)” abbreviates “Car is identified by RegNr”.

If an entity type has a compound reference scheme, this may be shown explicitly using an external uniqueness constraint (circled “u”). In our example, a City is identified by combining its Cityname and State (which in turn is identified by combining its Statecode and Country). For instance, I live in a city called “Bellevue” that is in Washington state; this state is in the country named “USA” and has the statecode “WA” (whereas Western Australia is in the country named “Australia” and has the statecode “WA”). Value types have dotted ellipses (e.g. “Statecode”), and a “+” indicates numeric reference.

A predicate is a sentence with holes in it for object-terms. 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): USbranch achieved Rank in Year. Predicates may have any arity (number of roles) and may be written in mixfix form (i.e. the object holes may be mixed in at any position within the predicate– this is essential for languages like Japanese where the verb comes at the end). 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 1 shows forward and inverse readings (separated by “ / ” if needed) for some of the relationships.

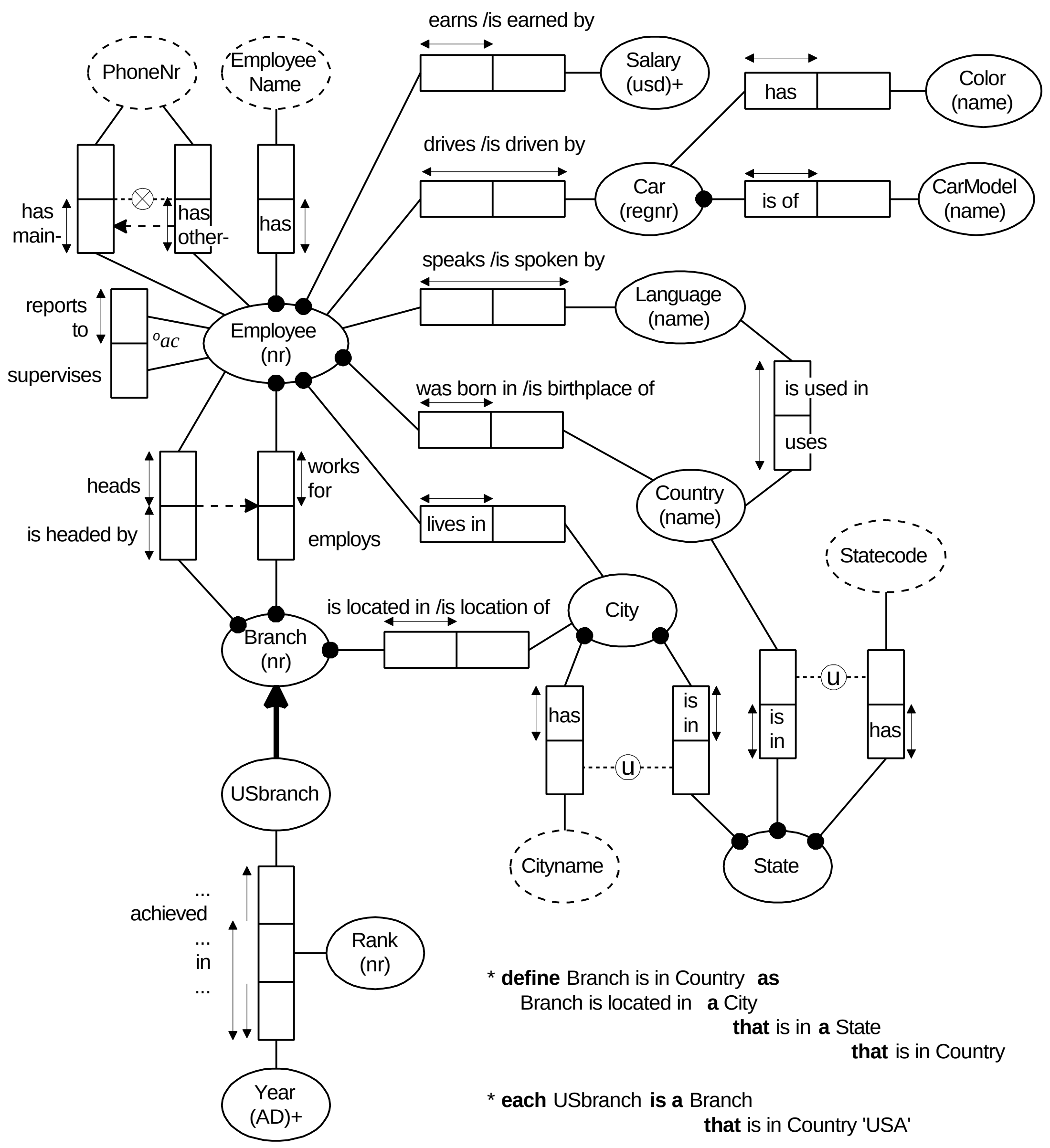

FIGURE 1: A sample ORM conceptual schema.

An arrow-tipped bar across a role sequence depicts an internal uniqueness constraint. For example, each employee has at most branch, but the same branch may employ more than one employee. The ternary has two uniqueness constraints: the right-hand one declares that a USbranch may achieve at most one rank in a given year; the left-hand one states that a given rank in a given year is achieved by at most one USbranch (i.e. no ties).

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). Subtypes are connected to their supertype(s) by arrows, and given formal definitions. Here we have only one subtype (USbranch). The two asterisked rules at the bottom of the figure declare a derived fact type, and a subtype definition: these textual rules are essentially ConQuer queries.

The circled “X” in the top right corner is a pair-exclusion constraint (an employee’s main phone number must differ from his/her other phone number). The dotted arrow just below the exclusion constraint is a simple subset constraint (if an employee has another phone number, he/she must have a main phone number). The dotted arrow from the heads predicate to the works-for predicate is a pair-subset constraint (each employee who heads a branch also works for that branch). The “°ac” constraint on the reporting relationship indicates this relationship is acyclic (no loops back to itself). ORM has other kinds of constraint not shown here. InfoModeler’s verbalization ability allows schemas to be entered or output in English sentences, so that it is not necessary to understand the diagram notation.

Sample ConQuer queries and SQL mapping

Although ConQuer queries are based on ORM, users don’t need to be familiar with ORM or its notation. A ConQuer query is set out 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.

With ActiveQuery, a 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 (this Point-to-point query feature is ignored for the remainder of this article).

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: List each employee who lives in the city that is the location of branch 52. This may be set out in ConQuer thus:

```
Q1  ✓Employee
          +-lives in City
                     +-  is location of Branch 52
```

This implicit form of the query may be expanded to reveal the reference schemes (e.g. EmployeeNr, BranchNr), and an equals sign may be included before “52”. For most users, the meaning of a ConQuer query should be clear enough (*semantic clarity*). ActiveQuery also generates an English verbalization of the query in case there is any doubt.

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. Notice how City is used as a join object type for this query. If attributes were used instead, we would typically have to formulate this is a more cumbersome way. If composite attributes are allowed we might use: List Employee.employeenr where Employee.city = Branch.city and Branch.branchnr = 52. If not, we might resort to: List Employee.employeenr where Employee.cityname = Branch.cityname and Employee.statecode = Branch.statecode and Employee.country = Branch.country and Branch.branchnr = 52. Apart from awkwardness, both of these attribute-based approaches violate the principle of *semantic relevance*. Since the identification scheme of City is not relevant to the question, the user should not be forced to deal explicitly with these details.

Even if we had a tool that allowed us to formulate queries directly in ER or OO models, and this tool displayed the attributes of the current object type for possible assimilation into the query (similar to the way Active query displays the roles of the highlighted object type), this would not expose immediate connections in the way that ORM does. For example, inspecting Employee.city does not tell us that there is some connection to Branch.city. The only way to do this is to use the domains themselves as a basis for connectedness, and this is one of the distinguishing features of ORM.

To illustrate other features of the query technology, it will help to show some SQL code that can be automatically generated from ConQuer queries. Using the Rmap algorithm [5], our conceptual schema maps to the relational schema shown in Figure 2 (for simplicity, domains are omitted). SQL queries apply to this.

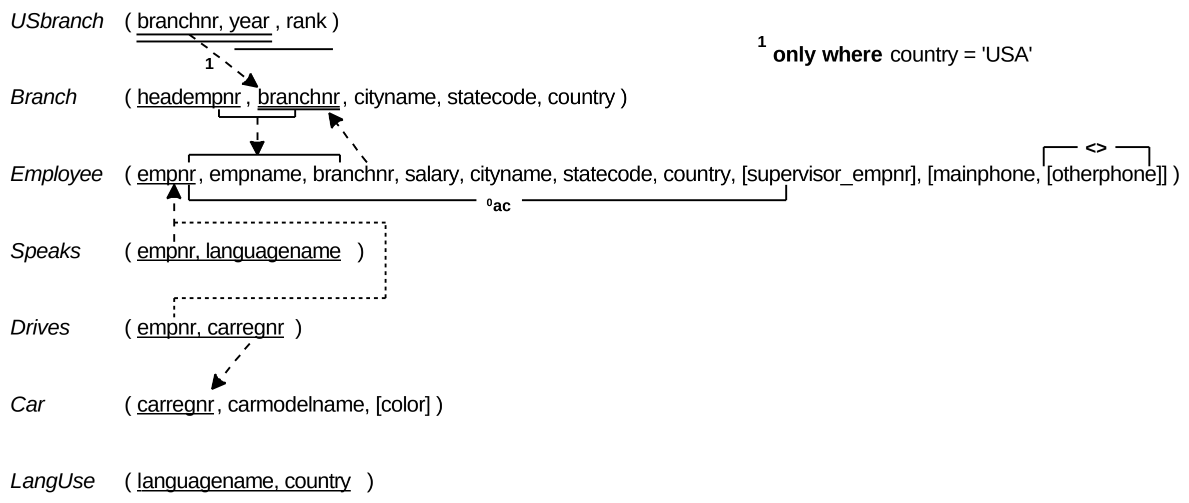
USbranch (branchnr, year , rank)
1 only where country = 'USA'
1
Branch (heademp nr , branchnr , cityname, statecode, country)
Employee (empnr , empname, branchnr, salary, cityname, statecode, country, [supervisor_empnr], [mainphone, [otherphone]])
<>
0ac
Speaks (empnr, languagename)
Drives (empnr, carregnr)
Car (carregnr , carmodelname, [color])
LangUse (languagename, country)

FIGURE 2: The relational schema mapped from the ORM schema of Figure 1.

In Figure 2, keys and uniqueness constraints are indicated by underlining (primary keys are doubly underlined where alternate keys exist). Optional columns are shown in square brackets. A subset constraint (e.g. foreign key constraint) is shown as a dotted arrow. Here the numbered qualification 1 enforces the subtype definition. For more about subtyping in ORM, see [6].

ActiveQuery maps ConQuer queries to SQL for a variety of DBMSs, in the process performing semantic optimization where possible by accessing the constraints in the ORM schema. SQL code for query Q1 is shown below (S1). Notice how the conceptual query shields the user from details about City's composite reference scheme. Moving through an object type corresponds to a conceptual join. Here the relational join is a result of the same city playing two roles that map to separate tables, with no foreign key connection. In contrast to this 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
        from  Employee as X1, Branch as X2
        where X1.cityname = X2.cityname
            and X1.statecode = X2.statecode
            and X1.country = X2.country
            and X2.branchnr = 52
```

ORM makes no use of attributes in base models. 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 example, suppose that after storing the previous query, we change the schema to allow an employee to live in more than one city (e.g. a contractor might live in two cities). The uniqueness constraint on Employee lives in City is now weakened, so that this fact type is now many:many. With most versions of ER, this means the fact can no longer be modeled as an attribute of Employee.

Moreover, suppose that we now decide to record the population of cities. In ER or OO this would require that City be remodeled as an entity type instead of as an attribute. Hence an ER or OO based query would need to be reformulated. With ORM based queries however, the original query can still be used, since changing a constraint or adding a new fact type has no impact on it. Of course, the SQL generated by the ORM query may well differ with the new schema, but the meaning of the query is unchanged.

As an even simpler example, suppose we wanted to list employee drivers and their branches. In ConQuer we have:

```
Q2  ✓Employee
        +-  drives Car
              +-  works for ✓Branch
```

With our current schema, employees may drive many cars but work for at most one branch. So the information is spread over two relational tables, and the SQL code is:

```
S2a    select X1.empnr, X1.branchnr
       from Employee as X1, Drives as X2
       where X1.empnr = X2.empnr
```

However suppose that in an earlier version of our schema, employees could drive at most one car. In that case, all the information is in one table and the SQL code is:

```
S2b    select X1.empnr, X1.branchnr
     from Employee as X1
     where X1.carregnr is not null
```

Not only is this code sub-conceptual (null values are an implementation detail) but it is unstable, since a simple change to a conceptual constraint on the driving relationship requires the code to be changed as well. If we now relaxed our schema to allow employees to work for more than one branch (e.g. contract employees) the SQL code would need to be changed again since an extra table is needed to store the works relationship. In all these cases, the ConQuer query stays valid: all that changes is the SQL code that gets automatically generated from the query.

An OO query approach is often more problematic than an ER or relational approach, because there are many extra choices on how facts are grouped into structures, and the user is exposed to these structures in order to ask a question. Moreover these structures may change drastically to maintain performance as the business application evolves.

In the real world, changes often occur to an application, and work is required to cater for the changes in the database structures. Even more work is required to modify the code for stored queries. If we are working at the logical level, the maintenance effort can be very significant. We can minimize the impact of change to both models and queries by working in ORM at the conceptual level and letting a tool look after the lower level transformations.

The simple examples above illustrate how ConQuer achieves semantic clarity, relevance and stability. Let's look briefly at its semantic strength (expressibility). Further details on this may be found in [2]. The language supports the usual comparators (=, <, in, like etc.), logical operators (and, or, not), and bag functions (count, sum etc.), as well as a maybe operator for conceptual left outer joins. Subtype/supertype connections appear as "is" predicates.

Suppose we want to list the branch number of any USbranch that did not achieve the top rating before the year 1998, as well as the name, and cars (if any) of the branch's head. This may be formulated in ConQuer thus:

```
Q3  ✓USbranch
          +-  **not** achieved Rank = 1 in Year < 1998
                  +-  **is** Branch
                           +-  is headed by Employee
          +-  has ✓EmployeeName
                  +-  **maybe** drives ✓Car
```

Notice how easy this is, especially if the tool provides the predicates for each object type. As a minor point, ActiveQuery currently uses the syntactical variant “possibly” for “maybe”. You are invited to provide the lengthy SQL for this query. Although straightforward, notice how you need to locate the relevant four tables and then decide on what columns to join, what kinds of join to perform (inner or outer) and then add the intra-table restrictions. In other words, to do this in SQL you need to worry about low level implementation details.

The most powerful feature of ConQuer is its ability to perform *correlations* of arbitrary complexity. As a simple correlation example, consider the query: Who supervises an employee who lives in the same city as the supervisor but was born in a different country from the supervisor? Here is one way of expressing the query in ConQuer:

```
Q4  ✓Employee₁
+-  lives in City₁
+-  was born in Country₁
+-  supervises Employee₂
                     +-  lives in City₁
                     +-  was born in Country₂ <> Country₁
```

When an object type appears more than once, ActiveQuery automatically appends subscripts to distinguish the occurrences. You can assert that the instances are equal by equating the subscripts (e.g. City_1). More generally, you can use comparators to compare instances (e.g. Country_2 <> Country_1). Try this in SQL. It’s not that hard, but you have to admit it’s easier in ConQuer!

As a harder example that includes a function as well as nasty correlation, 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. Recall that 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). This is quite tricky, and even experienced SQL users might get it wrong.

```
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)
```

Last year I taught advanced SQL to a group of 4th year university students who already had years of experience with SQL. I then gave them a simple introduction to ConQuer, followed by a list of varied questions in English that they had to translate into both ConQuer and SQL. Even without tool support, they had little trouble with the ConQuer formulations, but they experienced great difficulty with the SQL. I admit the SQL questions were pretty nasty (lots of correlated subqueries and functions), but I set a wide range of questions without trying to bias them in favor of either language. At any rate, the relative performance was so dramatic that it reinforced my impression that ConQuer is much easier to learn than SQL. Of course, more extensive trials are needed for a reliable empirical evaluation of the language.

Business Rules

Although ORM's graphical notation can capture more constraints than popular ER notations such as IDEF1X and UML class diagrams, it still needs to be supplemented by a textual language to provide a complete coverage of the kinds of constraints and derivation rules found in business applications. Research is currently under way to adapt ConQuer for this purpose. This is analogous to the way that SQL is used for formulation of queries as well as declaration of constraints (e.g. check clauses) and derivation rules (e.g. view declarations). All of ORM's graphical constraints can be verbalized in FORML, an ORM language that was designed specifically for this purpose. The ConQuer language is more general, can be used to define other business rules, and can be mapped automatically to SQL. Hence it could be used as a very high level language for capturing business rules in general.

As a trivial example of a derivation rule, Figure 1 includes a definition of the derived fact type: Branch is in Country. Now that you have some familiarity with ConQuer, you will recognize this definition as a ConQuer query. ActiveQuery allows you to define derived predicates, and store these definitions. These derived predicates (or “macros”) can then be used just like base predicates in other queries. Figure 1 also includes a subtype definition for USbranch. A subtype may be thought of as a derived object type, with its definition provided by a ConQuer query. Note that the subtype definition for USbranch made use of a derived predicate.

A constraint may be viewed as a check that a query searching for a violation of the constraint returns the null set. Hence constraints may also be expressed in terms of queries. Various high level constructs can be provided in the language to make it more natural than the not-exists-check-query form provided in SQL. Although there is no room here to go into detail, it should be clear that this approach is quite powerful.

Conclusion and Acknowledgement

This article outlined the benefits of lifting queries to the conceptual level, and argued that a truly conceptual query language should provide semantic strength, clarity, relevance and stability. It then indicated that ORM-based languages are especially suited for meeting these criteria, and gave examples of how queries can be formulated in one such language, ConQuer, and then mapped to SQL. Finally, it was noted that a conceptual query language can be used not just for queries but also for the declaration of business rules.

The ActiveQuery tool was constructed by a team of talented developers, now working at Visio Corporation. The fundamental research on the design of ConQuer and the associated mapping to SQL was performed jointly by Dr Anthony Bloesch and myself, and parts of this article are based on two of our papers [1, 2], in which a more formal discussion of the language’s semantics is provided.

References

1. Bloesch, A.C. & Halpin, T.A. 1996, ‘ConQuer: a conceptual query language’, *Proc. ER’96: 15th Int. Conf. on conceptual modeling*, Springer LNCS, no. 1157, pp. 121-33.

2. Bloesch, A.C. & Halpin, T.A. 1997, ‘Conceptual queries using ConQuer-II’, *Proc. ER’97: 16th Int. Conf. on conceptual modeling*, Springer LNCS 1131, pp. 113-26.

3. Cattell, R.G.G. & Barry, D. K. (eds) 1997, *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, San Francisco CA (see ch. 4 for a definition of OQL).

4. 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.

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

6. Halpin. T.A. 1995, ‘Subtyping: conceptual and logical issues’, *Data Base Newsletter*, ed. R.G. Ross, Database Research Group Inc., vol. 23, no. 6, pp. 3-9.

7. Halpin, T.A. 1996, 'Business rules and Object Role modeling', *Database Programming and Design*, vol. 9, no. 10 (Oct. 1996), pp. 66-72.

8. 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.

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