A Simple Metamodel for Fact-Based Queries

Clifford Heath

Data Constellation
http://dataconstellation.com

Abstract. Fact-based models are built by expressing elementary facts using natural verbalizations. By generalizing individual objects to object types, facts to fact types, and adding constraints, a schema for any domain can be constructed.

Fact-based schemas have many advantages, including being highly amenable to construction of natural verbalizations, since they were originally derived from such verbalizations.

However, it is not common practice to consider queries during modeling, so only limited attention has been paid to how to model them. Queries are not only useful for extracting data, but also to express complex business constraints. An effective meta-model for fact-based queries is presented, as an extension of a tiny subset of the meta-model of Object Role Modeling. Examples expressed in the Constellation Query Language show how to populate the query meta-model.

1 Fact Based Models

In a fact-based model, an object type may be either:

- a data type - a set of values (potentially infinite) for which there exists a canonical written (lexical) form;
- a value type - a type relevant to the domain of discourse and which maps to a data type (a meaningful thing you can write down, like a name); or
- an entity type, of which each instance is uniquely identified by the combination of one or more roles played (in so-called existential fact types) by that instance.

Each fact asserts some relationship between objects, or a characteristic or self-relationship of one object, and is an instance of a single fact type. Each fact type allows the assertion of the predicate (which describes the relationship or characteristic) over objects of certain specified object types or their sub-types. Some or all instance facst of a given fact type may be derived using a query. Fact-based models also include various kinds of constraints, but those are not considered further here.

In addition to the object types and fact types, a fact-based model also contains a population of objects and facts, which are the subject of queries.

Y.T. Demey and H. Panetto (Eds.): OTM 2013 Workshops, LNCS 8186, pp. 534–544, 2013.

2 Approaches to Fact-Based Queries

A fact-based model contains a population of object instances and elementary fact assertions. A query over such a model asks whether the population contains certain objects related to one another (or not) by specified facts. Although querying of fact-based information models has been a consideration since the inception of such models, the approaches have seldom been adequately explored, and very little has been published. One of the first fact-based modeling languages, RIDL[2] included a powerful query language, but it is unknown whether an implementation of this query language was ever completed. Instead, papers say "the meaning should be self-evident" and suggests that manual effort is required to write a program to evaluate the query[3]. Many researchers have treated Fact Based Modeling (FBM) as a means to design a conventional relational database, and relegate querying to the existing relational query tools such as SQL, as in CLCE[1], or refer to other work in mapping the query to first-order logic and thence to PROLOG, for example. However, mapping a fact-based query to a relational query is not a well-defined operation unless the original query is well-defined, and neither the mapped query nor the relation which results from processing the query can easily be represented in the terms of the original fact-based model unless the reverse mapping is also well-defined. Halpin[6] et al implemented ConQuer, which has multiple presentation modes including purely textual and tree-structured text with graphical annotations. Examples of ConQuery show its power, and the papers explain that the semantics is based on the domain relational calculus (or alternatively, a first order logic), but this author is unaware of the publication of a complete language specification or its metamodel. Query execution is by translation to SQL, with several good examples provided, but the results are not represented or verbalised using the original fact model, which could cause difficulties in model evolution due to exposing the underlying relational mapping and/or identification schemes. ACE[5] has been the focus of various experiments in synthesising programs and in theorem proving. Kuhn[4] defines his query language by defining its grammar; its semantics is defined only in terms of a parser implementation that maps it to PROLOG, and he does not elaborate further. Within the more limited framework of the semantic web (open-world binary fact types represented as RDF triples) SPARQL[8] provides a query language, and there exist reasoners/inferencers such as RacerPro[9] which execute such queries. Queries are also present as derived fact types within NORMA[10] Pro, but this is not yet publicly available.

The goal of this paper is to present a metamodel for queries as an extension to the metamodel of ORM, so that a verbal or graphical query can be compiled (conceptualised) into a fact population using the query metamodel. This compiled form is capable of being re-verbalised into equivalent (though not necessarily identical) source text. Both these operations have been implemented (excluding full support for aggregates and alternation) in the English versions of the Constellation Query Language[7] and this implementation is publicly available. A French version currently supports compilation, but not yet re-verbalisation, and work on Mandarin support is underway. Evaluation of such queries is an ongoing subject of work not discussed in this paper.

3 Query Model

The ORM2 diagram in Figure 1 shows the core features of fact-based models (Object Type, Role, Fact Type, omitting their reference schemes) across the top, then it introduces the concept of a Query, that contains Variables which may be satisfied by objects of a specific type, subject to certain conditions. The rest of this paper discusses the conditions under which a query is satisfied. The queries used for the examples are shown using the Constellation Query Language, though other fact-based query languages could also be used.

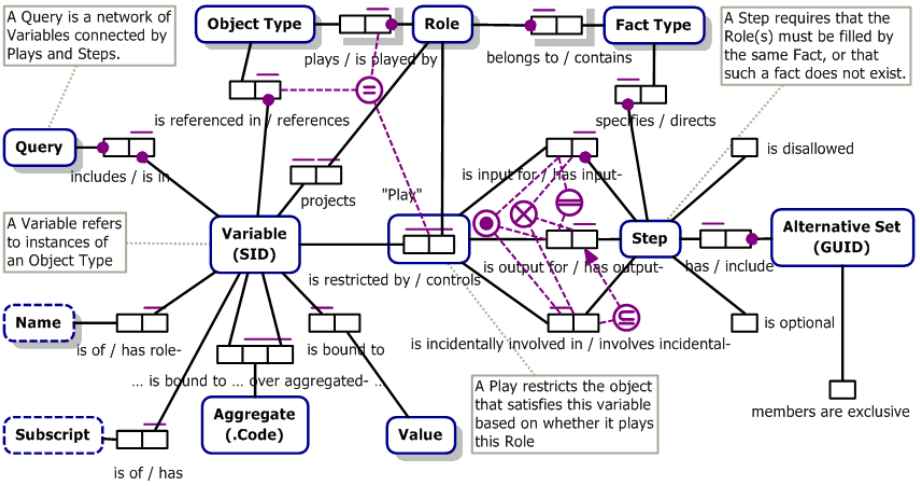

Fig. 1. Query Metamodel

Some of the variables in the query may be bound to particular values (hence are individually satisfied if those values exist), while others are bound only (via Play and Step) to other variables. It is not interesting to consider queries which contain disjoint graphs, since the result is just the cartesian product of the results of the separate subqueries. Although a Variable may involve many connections, we do not use n-ary steps, rather we model the query as a di-graph. The direction of an arc (a Step) only matters in the case of outer joins (optional steps), but we retain the notion of an input Play and an output Play for each Step. Because a step may traverse an n-ary fact type, the extra roles are each assigned to a Play with an associated Variable; these incidental Plays are not conceptually distinct from output Plays, but are a vestige of the current implementation. We now begin the full exposition of the meta-model.

4 Variables

Each query contains at least one variable. Each variable references one object type. When more than one variable in a query references the same object type, either the optional role-Name or Subscript must be populated to prevent ambiguity in the CQL verbalisation.

In satisfying a query, each variable may only be paired with an object (drawn from the model's population) of the respective object type, or in the case of an object type which is a data type, with a value which is included in that data type's value set (regardless of whether that value occurs in the model's population).

The query is satisfied when all variables are correctly paired and all other query conditions (described later) are met. Query results include the values of data- or value-types, and the identifying role values for entity types. Additional values linked to the projected objects may also be returned, but the details are not covered in this paper.

Assuming a data type, a value type, and an entity type called respectively Integer, Name, and Person, the following CQL queries containing one variable will be satisfied by any Integer (an infinite set, or as defined by the data type), any Name that exists in the model's population, and any Person that exists in the model's population:

```
Integer?
Name?
Person?
```

Simple conjunction of multiple variables with no other restriction is satisfied by any combination of the variables:

```
Performer, Venue?
```

is satisfied by any combination (existing in the population) of Performer and Venue. As noted, any such query containing disjoint sub-queries returns the cartesian product.

The remaining features of the query meta-model provide different ways to restrict the combinations of variable values that satisfy a query.

5 Query Conditions

5.1 Bound Variables

A variable may be bound to a specific value if that variable references an object type that maps to a single value. Allowable object types are any data type, any value type, or an entity type having a single identifying role (i.e. played by a bindable object type). A variable which references an entity type that has more than one identifying role, or whose single identifying role is played by such an entity (transitively) may not be bound to a value. Steps over the entity's existential facts must instead be provided to select the compound identifier. However, a proposed short-hand feature for CQL would allow definition (using an extended regular expression with named capture groups) of a syntax for a single literal to be parsed into the required multiplicity of identifying values, allowing (for example) given-name and family-name to be extracted from either "Heath, Clifford" or "Clifford Heath".

A bound variable that references a data type is satisfied only by that value (which must be valid for the data type). A bound variable that references a value type is only satisfied by the instance of that value type (populated in the model) which maps to that value. A bound variable that references an entity type is only satisfied by the instance of that entity type (populated in the model) which is transitively identified by that value.

As an example if Person is identified by the (value type) Name, the following query is satisfied only if the respective Person exists in the model's population:

```
Person 'Daniel'?
```

This query asks the question "*Does there exist a Person named Daniel?*"

5.2 Role Playing (Plays)

Often, each object type that is referenced by a variable will be expected to play one or more roles (or possibly, *not* to play a possible role, but more on that later). Each such possible case of role playing is indicated by a **Play** instance. Each Play instance must be linked via a **Step** instance to the fact type that contains that role. The details surrounding the Step restrict the variable to a subset of the otherwise available objects that might have satisfied the query.

For example, in a model that has the unary fact type "*Person smokes"*, only a Person who smokes satisfies the following CQL query:

```
Person smokes?
```

In this case there is a Play that indicates that it is relevant whether or not the person smokes (a role they can play). The Play instance is linked to a Step that specifies the unary fact type, so that the query can only be satisfied by a Person who smokes.

Fact Conditions (Steps)

The case of a unary step has been covered, but more commonly, two variables may be restricted to objects playing roles of the same fact instance. We say that the two Plays (one input Play, one output Play) are connected by a **Step**. This query model does not directly represent that three or more roles must be in the same fact – that is made possible by adding Steps over a shared Play.

In the case of an n-ary fact type (more than two roles), the fact that satisfies the step will have additional roles, which may be represented as incidental Plays. These are included in the meta-model for the sake of completeness, though it's not required that they play any other part in the query. The variables associated with the incidental Plays provide the instances that complete the facts for the respective steps.

A Step may be one of a set of **alternates**, may be **disallowed** or may be **optional**, as discussed in following sections.

In a model where perhaps *"Person has birth Date"*, then only a Person born on the specified date satisfies the following query:

```
Person was born on birth Date '20/07/1987'?
```

This query contains two variables, two plays (the roles *Person* has a birth-date, *Date* is birth-date of some person) and one step (over *"… was born on birth-…"*).

Any Person who smokes and whose birth-date is known satisfies the following query:

```
Person smokes and was born on birth Date?
```

Here Date has one Play and Person has two (one for possibly being a smoker, and one for possibly having a recorded birth date). The two steps require that the person actually be a smoker (this step has no output play), and to link the person with their birth date.

5.3 Disallowed Steps

Sometimes it's necessary to require that a matching fact does not exist in the population. In this case the unary "Step is disallowed" is asserted. Any instance of a fact disallowed by a Step disqualifies the role players from satisfying those respective Plays in the query.

If we have a model containing "*Person attended Party"* and "*Person was invited to Party"*, we could ask the following:

```
Person attended Party and was not invited to?
```

This verbalization contains two separate contractions (both are optional) and an implicitly-negated fact type reading. It is short for:

```
which Person attended which Party and it is not the case
that that Person was invited to that Party?
```

Again, there are two variables, but here there are four plays and two steps, one of which is negated. The left- and right-contractions above involve the same players as in the previous clause, and each is allowed only when the player is in the same end position in both predicates. In some situations, only one or no contraction is possible.

Implicit negation applies when a reading can only be matched by dropping the word "*not*" from any position in the clause. This allows very non-English expressions, and should only be used where an explicit negative reading has not been provided. The negation prefix "*it is not the case that*" may be used before any reading, and is preferred for unary fact types. Better than either is the new capability to define explicit negative readings for any fact type, such as *"Person doesn't smoke*, *Person is a non-smoker"* as negative readings for *"Person smokes"*.

Here, the earlier formulations are preferable to the later ones:

```
which Person is a non-smoker?
it is not the case that Person smokes?
which Person smokes not?
```

Explicit negative readings are preferable in most instances, not least because the limited linguistic tools available cannot correctly re-insert implicit negations when re-verbalising a query.

5.4 Optional Steps

Sometimes a step may be optional - taken if a matching fact exists but ignored otherwise. This allows the relational equivalent of an outer join. The step is simply marked optional. For example, any Person who smokes satisfies the following query, and we also learn their birth Date **only** if that is recorded.

```
Person smokes and maybe was born on birth Date?
```

In this case it may happen that a Variable that is only reached by the output side of optional steps (birth Date in this case) is not populated; the query can still be satisfied. In the above query, a smoker may be identified without a birth date. This is the only sense in which a Step is directional (having an input side that is not interchangeable with its output side).

A variable that is not necessarily populated may be projected into a derived fact type, but the incompletely populated results of the projection are omitted, otherwise the derived fact instances would be incomplete.

5.5 Alternate Steps

When there are alternate steps that may be taken, each step is implicitly optional, but at least one step must be taken (or exactly one, if the steps are exclusive). The alternative steps are collected into an Alternative Set. In CQL, exclusive steps are expressed using *"but not both"* key phrase (or *"but only one"* if there are more than two alternatives) after all the alternatives.

```
some Person may attend some Party where
    that Person was invited to that Party or
   that Party is not invitation-only;
```

The syntax of CQL allows mixing conjunction (*and* or *comma*) with alternation (*or*) but only with a fixed precedence and no parentheses (apart from the use of parentheses in aggregates, see below). It does not allow arbitrary combinations of these conjunctions. The *and* operator has highest precedence, followed by *or*, followed by ',' (comma). More complex nesting of Boolean conditions can only be expressed using a derived fact type (which will be represented as a nested query).

6 Special Fact Type Steps

6.1 Subtyping

When an object type is a subtype, there is an implicit identity fact type for the subtyping. In CQL, these identity fact types have the predicate *"is a kind of/is a"*, so they

can be explicitly invoked in query steps. However, in the case where a query invokes a predicate with an object type that is a sub-type or super-type of the expected type, an implicit subtyping Step (over the identity fact type) is created in order to complete the query. This subtyping step may be optionally included or omitted from a subsequent re-verbalization, as long as omitting it creates no ambiguity.

6.2 Objectification Steps

For each role of an objectified fact type, there is an implicit binary "link" fact type that relates the role player to the objectifying instance, as exemplified here:

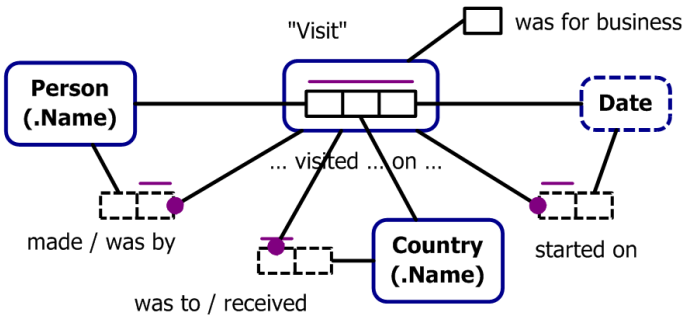

Fig. 2. Example of objectification link fact types

In CQL, these link fact types (for which the readings shown cannot yet be specified) can be traversed in an objectification step using special syntax, as exemplified here:

```
Visit (in which Person 'James' visited Country on Date)
was for business?
```

This syntax creates a step over the respective link fact type, which completes the query asking *"when and where did James travel for business?"* The unbound variables that must be populated to satisfy the query are Visit, Country and Date. Even when the objectifying entity type is not present in the query, a step through the objectified fact type is represented using two such objectification steps and a variable is created for the objectification. The re-verbaliser can elide this extra variable if it plays no other role in the query.

6.3 Arithmetic Steps

Arithmetic facts may also be included in queries. For example, addition is a ternary fact type ($a = b + c$, or alternate readings $b + c = a$, $a - c = b$, etc.) These fact types come from the mathematics of the underlying data types, so the same patterns reoccur for each numeric data type. In CQL, any role that maps to a single value may be used to invoke them. It's perfectly valid to ask:

```
Integer 1 + Integer 1 = Integer 2?
```

As another example, assuming the system value instance *"Now"*, and the date fact type *"Date(1) + Days = Date(2)"* where Days is an Integer defined in units of 1 day, and our model contains *"Person was born on birth- Date"*, we might ask:

```
Person was born on birth Date <= Now - 21 years?
```

Any Person who was born at least 21 years ago satisfies this query. The required approximate conversion from years to days is provided by the *units* subsystem in CQL. This subsystem also provides for exact (rational), accurate (floating point), approximate or ephemeral conversions between 500 different standard units, and has a syntax for user-provided conversions. Conversion functions can be generated between arbitrary values by algebraic manipulation through fundamental units. Ephemeral conversions are provided by an outside source, such as the conversion rate on a specific day from Australian to USA dollars. The details of the units model are yet to be published, but are largely derived from existing work.

6.4 Data Type Conversion Steps

Where incompatible data types are used to populate a step (or roles that map to such data types), appropriate type conversion steps may be automatically provided. These step over data type conversion fact types, which are binary fact types having the two data types as players. The implicit steps should not be verbalized, but whether or not to populate them explicitly is a decision for the implementer.

6.5 Identification or Mapping Steps

When a role that requires a data type is populated with a value type or an entity type having a single identifying value, the required fact type step over the existential fact type is implied, but is not populated into the query model. Similarly when a value type is required but a singly-identified entity type is provided, the step which specifies the identifying fact type(s) is elided from the query model. This is an implementation choice which does not affect the interpretation.

7 Projection and Derived Fact Types

It's not necessarily the case that we're interested in the values that satisfy **all** of the unbound variables of a query. What we want instead is to project the distinct combinations of values for one or more variables, while ignoring the values that satisfy other variables. The other values must exist for the query to be satisfied, but we don't want to know the details. In CQL, this is supported using the *which* and *some* qualifiers in the query:

```
which Person came to some Party and was not invited to?
```

Here we don't care about the actual party, just the fact that the person is a gate-crasher. Note that "some" here refers to the existence of some particular Party as required to satisfy the fact type. CQL never refers to collections, instead it defines rules about individual objects which may be used to derive collections of suitable objects. In the same way, multiple individual people may satisfy the query, but the query refers to each one individually. This avoids the ambiguity of plural verbalisations.

Alternatively, we may wish to use such a query to define a new derived fact type:

```
Person is a gate-crasher where
   that Person came to some Party and was not invited to;
```

Person is projected from the query to define a role in a new unary fact type. If the same person has gate-crashed more than one party, the single fact remains that he or she is a gate-crasher – the individual instances are lost. A variable is projected into a derived fact type by populating the query meta-model with the fact *"Variable provides projected- Role"*. A query may only project variables into a single derived fact type. To construct a query that requires multiple levels of projection requires the use of derived fact types, since projection only occurs on the result of each sub-query.

8 Aggregates

An aggregate summarizes a single variable projected from a sub-query. Each aggregating function is identified by a code-name, such as "sum", "count", "average", "stdev", etc. Here's an example of the use of an aggregate in a derived fact type:

```
Company pays SalaryBill where
    Company employs Person and
    SalaryBill = sum of Salary in (Person receives Salary);
```

Note that the sub-query may include variables that occur only in the sub-query, as well as variables that occur also in the enclosing query, but not variables that occur in other queries. The CQL parser allows any aggregate function name to be used, which allows custom implementations to easily be added.

9 Conclusion

A simple yet powerful meta-model has been presented which covers a large class of queries of interest across a wide range of domains. The model is straightforward to correctly populate and to verify the validity of query results, and supports effective reverbalisation of the compiled query into equivalent source text. Furthermore, the results that satisfy such a query can also be verbalised into complete or relative answer sentences, though this has not yet been implemented in CQL.

References

1. Sowa, J.: Common Logic Controlled English, http://www.jfsowa.com/clce/specs.htm (February 24, 2004, downloaded May 2013)
2. Meersman, R.: The RIDL conceptual language. In: Research Report, Int. Centre for Information Analysis Services, Control Data Belgium, Brussels, Belgium (1982)
3. Meersman, R., Van Assche, F.: Modeling and manipulating production data bases in terms of semantic nets. In: Proceedings IJCAI 1983 Proceedings of the Eighth International Joint Conference on Artificial Intelligence (1983)
4. Kuhn, T.: Controlled English for Knowledge Representation. PhD thesis, Faculty of Economics, Business Administration and Information Technology of the University of Zurich (2009)
5. Fuchs, N.E., Schwertel, U., Schwitter, R.: Attempto Controlled English (ACE) Language Manual Version 3.0. Technical Report, University of Zurich (1999)
6. Halpin, T.A., Bloesch, A.C.: Conceptual Queries Using ConQuer-II. In: Embley, D.W. (ed.) ER 1997. LNCS, vol. 1331, pp. 113–126. Springer, Heidelberg (1997)
7. Heath, C.: The Constellation Query Language. In: Meersman, R., Herrero, P., Dillon, T. (eds.) OTM 2009 Workshops. LNCS, vol. 5872, pp. 682–691. Springer, Heidelberg (2009)
8. The World Wide Web Consortium, SPARQL Query Language, W3C Recommendtion, http://www.w3.org/TR/sparql11-query/
9. Racer Systems, downloaded, Racer Pro 2.0: A system overview (May 2013), http://www.racer-systems.com/products/racerpro/preview/overview.phtml
10. Halpin, T., Curland, M.: NORMA, ORM Solutions Limited, http://www.ormfoundation.org/files/folders/norma_the_software/default.aspx