Abstract

In this paper, we propose a novel methodology and architecture that enables system designers to support domain-specific conceptual query languages (DSC-QL) with dynamic support for domain-specific functions (DSFs). This methodology also specifies how to create a domain-specific conceptual data model (DSC-DM) that consists of a data structure diagram, definition of DSFs, and two special tables for additional domain semantics. DSC-QL is built on DSC-DM and enables users to query their data in the terms of concepts and functions that are commonly used in their domain. It is DBMS-independent and can be translated into SQL, OQL, XQuery, or other query languages. The methodology is applicable in any domain and we present a case study in the neuroscience domain.

1. Introduction

Currently, research in life sciences is becoming more and more information intensive. Large amounts of data are being accumulated in various formats at the experimental level and it is becoming more and more complex and difficult to manage. Traditional data storage methods such as notebooks and literature are no longer sufficient and being replaced by various databases. As a consequence, end-users are not able to directly interact with their data in repositories. They have to either ask database administrators to retrieve the data for them, or use pre-developed form-based query interfaces to get the data by themselves. In order to enable users to directly query their data in terms of the concepts and functions that are commonly used in their domain,
we need query languages at the conceptual level (i.e. DBMS-independent) that support domain-specific functions. It is obvious that traditional query languages such as SQL and OQL do not meet the criteria because they are DBMS-dependent and require users to work explicitly with the data structure at the database level (see Figure 1). In addition, SQL and OQL do not have built-in support to the functions that are commonly accepted in a particular domain because they are designed to be domain-independent and primarily for database administrators and developers.

![Diagram: Domain-Specific Conceptual Query Language vs. Traditional Query Language](image)

**Figure 1:** Domain-Specific Conceptual Query Language vs. Traditional Query Language

In most applications, the query services to end-users are provided through form-based query interfaces. Although form-based query interfaces do not require users to know anything about the query language and the database details, they typically have limited expressive power and support only predefined types of queries. They are application-dependent, can rapidly become obsolete as the underlying database schema evolves, and have to be re-programmed.

In this paper, we propose a novel methodology and architecture that enables system designers to support domain-specific conceptual query languages (DSC-QL) with dynamic support for domain-specific functions (DSFs). This methodology also specifies how to create a domain-specific conceptual data model (DSC-DM) that consists of a data structure diagram, definition of DSFs, and two special tables for capturing additional domain semantics such as meta-information, annotation and the relationship between entities and attributes. DSC-QL relies on DSC-DM and provides conceptual-level querying capability. It supports all concepts and
functions defined in DSC-DM. We integrate it with some features of OQL like dot-path expressions. DSC-QL queries can be automatically translated into SQL, OQL, XQuery, or other query languages according to the type of underlying database system. It is DBMS-independent and does not require users to know the details of the database.

The proposed methodology can be applied to any domain. We show its application in the neuroscience domain to design and implement a domain-specific query language and database system.

2. System Architecture

The architecture of the system resulted from this methodology is shown in Figure 2.

![Figure 2: The architecture of domain-specific conceptual data modeling and querying system](image)

2.1 Domain-specific conceptual data model

DSC-DM consists of a data structure diagram, domain-specific functions, and two special tables for meta-information and annotation (Figure 4 shows an example of the DSC-DM data diagram in neuroscience domain). The notation in data structure diagram is similar to enhanced Entity Relationship data model (ER) [9] because it is straightforward and easy to understand for naïve users. We did try to use some more powerful data models such as UML (Unified Modeling
Language [3]) and ORM (Object Role Modeling [4]) in our previous bioinformatics projects, but we did not get positive response from the biologists. However, the expressive power in enhanced ER model sometimes limits the capture of some domain semantics such as meta-data and annotation. Therefore we designed two special tables (Figure 3): annotation table and meta-attribute table to capture following domain semantics:

i. Support the relationship between attributes and entities (such as annotation).
ii. Support controlled values for a certain attribute.
iii. Support data structures like Set and Enumeration.
iv. Allow attributes to have meta-attributes that are primarily used for the meta-information such as data type, unit, standard error, controlled values, and constraints.

![Figure 3: Annotation and Meta-Attribute table schemas](image)

Domain-specific functions (DSF) in DSC-DM are defined by domain experts and implemented by database developers. Their implementation details are stored in an XML file (DSF-XML) that will be dynamically read in by DSC-QL Translator at run time for query translation. As a result, DSF can be easily updated at any time without recoding the system.

When the database schema is generated from the DSC-DM, the entities and relationships in DSC-DM become the data sorts in DSC-QL, and are automatically mapped to the tables, classes, or elements in the databases. DSC-XML is an equivalent data model to DSC-DM, but in XML format. XML-Convertor converts DSC-DM into DSC-XML which is then used by DB-Generator to create a corresponding database schema. DB-Generator also generates the mapping from entities and relationships in DSC-DM to the tables/classes/elements in the database. This mapping will be used by DSC-QL Translator to translate a DSC-QL query statement into a DBMS-supported query statement (See Figure 2).
2.2 Domain-specific conceptual query language

DSC-QL relies on DSC-DM, which can be translated into DBMS-supported query languages like SQL, OQL, and XQuery. DSC-QL Translator handles the translation according to the information loaded from two XML files, “DSC-XML” and “Mapping from DSC-DM to DB schema”. This DBMS-independent feature in DSC-QL can facilitate the data integration in a federated system greatly. DSC-QL has very simple syntax and is readily usable, whose syntax, semantics and the translation algorithms to SQL are introduced in Section 3. DSC-QL automatically supports the concepts, relationships and domain-specific functions defined in DSC-DM. Through DSC-QL, end-users can query their data at the conceptual level in the terms of concepts and functions that are commonly used in their research. They do not have to know the details of the underlying database. DSQ-QL integrates the dot-path expression feature from OQL, which makes the users feel more comfortable than the SQL join command does.

3. Syntax and Semantics of DSC-QL

The prototype of DSC-QL syntax was initially proposed by biologists with whom we worked with in the past years. Here we use BNF notation to describe the syntax of DSC-QL. Two translations from DSC-QL to SQL (on Oracle 10g) and OQL (on EyeDB) have been implemented. The translation algorithm from DSC-QL to SQL is presented.

3.1 Syntax

```
<DSC-QL>::= "(" <result_list> "")" | "("<result_list> ")" "["<condition_list> "]"
,result_list>::= <result_term> | <result_term> "," <result_list>
,result_term>::= <declaration> | <dotExp_primitive>
declaration>::= <datasort> ID | <dotExp_class> ID
dotExpression>::= <dotExp_primitive> | <dotExp_class>
dotExp_primitive>::= <dotChain>"."PRIMITIVE_ATT
dotExp_class>::= <dotChain>"."CLASS_ATT
dotChain>::= ID | ID."" <dotChainTail>
dotChainTail>::= CLASS_ATT | CLASS_ATT "." <dotChainTail>
condition_list>::= <condition_term> | <condition_term> "," <condition_list>
```
3.2 Semantics

DSC-QL queries have a very simple structure consisting of a <result_list> enclosed in ( ) and a <condition_list> enclosed in [ ], i.e. (result_list)[condition_list]. There are two types of <result_term>s in <result_list>, 1) <declaration>: declare a data sort instance. A default attribute of the instance will be returned. For example, the declaration “datasortA a” means the ‘name’ (the default attribute specified in DSC-DM) of datasortA instances that satisfy the query conditions in <condition_list>; 2) <dotExp_primitive>: a primitive property of a data sort specified by the path expression. For example, the result term “a.datasortB.name” means the value of the primitive property ‘name’ of a datasortB instance associated with a datasortA instance ‘a’ satisfying the query conditions.

The <condition_term> in the <condition_list> can be a <declaration>, a comparison term, or a DS function. The <declaration> in <condition_list> does nothing but define a data sort instance in the query. The comparison terms and DS functions allow users to specify their query criteria. In addition, DS functions can also be used to declare an instance implicitly. For example, NS_F(a, b) implicitly declares two instances ‘a’ and ‘b’ (if they are not declared in the query) according to the input data sort types of DS function NS_F.

The dot-path expression in DSC-QL is similar to the path expression in OQL. The major difference is that the node in DSC-QL path expression can have a set data structure.

Let’s use RT₁, RT₂, …, RTₙ to represent the domain of each <result_term> in <result_list> respectively. Then the domain of the final result, RL, is the set of n-tuples, whose first element is
from RT₁, the second element is from RT₂, and so on. The query result includes the tuples in RL, which satisfy all query criteria in the condition list.

### 3.3 Translation algorithm from DSC-QL to SQL

**Input:** DSC-QL query statement

**Output:** SQL query statement

**Method:**
- Initialize a SQL query statement with empty SELECT, FROM, and WHERE clauses;
- Initialize sysID and increase sysID whenever it is used below;
- Scan the DSC-QL query statement from left to right;

**For each <result_term> enclosed in ( )**

If it is a <declaration>
   - **Case 1:** <datasort> ID, Append “<datasort> as ID” in FROM clause, and append “ID.name” in SELECT clause
   - **Case 2:** <dotExp_class> ID e.g. CID.<nodeA>.classB ID
     - Append “ID.name” into SELECT clause, and append following string into FROM clause
       \[
       \text{"(select classCID.pk as pk, sysID.name as name from classCID, nodeA, classB as sysID}
       \text{where classCID.pk = nodeA.fkToClassCID AND nodeA.pk = sysID.fkToNodeA) as ID"}
       \]
     - Append “AND CID.pk = ID.pk” into WHERE clause

If it is a <dotExp_primitive> e.g. CID.<nodeA>.<nodeB>.prop
   - Append “sysID.prop” into SELECT clause, and append following into FROM clause
     \[
     \text{"(select classCID.pk as pk, sysID1.prop as prop from classCID, nodeA, nodeB as sysID1}
     \text{where classCID.pk = nodeA.fkToClassCID AND nodeA.pk = nodeB.fkToNodeA) as sysID"}
     \]
   - Append “AND CID.pk = sysID.pk” into WHERE clause

**For each <condition_term> enclosed in [ ]**

If it is a <declaration>
   - **Case 1:** <datasort> ID, append “<datasort> as ID” into FROM clause
   - **Case 2:** <dotExp_class> ID e.g. CID.nodeA.classB ID
     - Append following string into FROM clause
       \[
       \text{"(select classCID.pk as pk, classB.* from classCID, nodeA, classB}
       \text{where classCID.pk = nodeA.fkToClassCID AND nodeA.pk = classB.fkToNodeA) as ID"}
       \]
     - Append “AND CID.pk = ID.pk” into WHERE clause

If it is NS_FUNCTION\(<\text{parameter_list}\>), Call NS_FUNCTION_HANDLER

If it is <operand> COMPARISON_OP <operand>
   - **Case 1:** both <operand>s are PRIMITIVE
     - Append “AND <operand> COMPARISON_OP <operand>” into WHERE clause
   - **Case 2:** only one <operand> is <dotExp_primitive> e.g. “CID.nodeA.nodeB.prop = 10”
     - Append following string into FROM clause
       \[
       \text{"(select CID.pk as pk from classCID, nodeA, nodeB where classCID.pk = nodeA.fkToClassCID}
       \text{AND nodeA.pk = nodeB.fkToNodeA AND nodeB.prop = 10) as sysID"}
       \]
     - Append “AND sysID.pk = CID.pk” into WHERE clause
   - **Case 3:** both <operand>s are <dotExp_primitive> e.g. CID.nodeA.prop = CID2.nodeB.prop
     - Append following string into FROM clause
       \[
       \text{"(select CID.pk as pk, CID2.pk as pk2 from classCID, nodeA, classCID2, nodeB}
       \text{where classCID.pk = nodeA.fkToClassCID AND classCID2.pk = nodeB.fkToClassCID2}
       \text{AND nodeA.prop = nodeB.prop) as sysID"}
       \]
     - Append “AND sysID.pk = CID.pk AND sysID.pk2 = CID2.pk” into WHERE clause

The translation algorithm from DSC-QL to SQL scans the input DSC-QL query from left to right, and outputs a corresponding SQL query. In general the \textit{SELECT} clause of SQL is resulted...
from the result terms, FROM clause is resulted from the declaration terms, path expressions and
NS functions, and the WHERE clause is from the comparison terms and DS functions. A special
function NS_FUNCTION_HANDLER takes care of the translation of each DS function, whose
basic idea is to create a sub-query in WHERE clause and associate it with other conditions. The
dot-path expression will be replaced by a series of SQL joins in a sub-query in the FROM clause.
For example DSC-QL statement, \texttt{CID.nodeA.nodeB.prop = 10} will be translated into following sub-
query in FROM clause: 
\begin{verbatim}
(SELECT classCID.pk AS pk FROM classCID, nodeA, nodeB WHERE classCID.pk = nodeA.fkToClassCID AND nodeA.pk = nodeB.fkToNodeA AND nodeB.prop = 10 ) as sysID_1
\end{verbatim}
and following condition in WHERE clause: \texttt{AND sysID_1.pk = CID.pk}
Section 4 gives three translation examples from a specific DSC-QL queries into SQL queries.

The translation algorithm from DSC-QL to OQL is similar to the algorithm to SQL. The
major difference is that the dot path expression is translated into multiple dot-path expressions in
order to remove the node with a set structure from the path. For example, the DSC-QL
comparison term: \texttt{id.NodeA.setNodeB.setNodeC.NodeD.prop > 10} will be translated into:

\begin{verbatim}
id.NodeA.setNodeB sysID_1, sysID_1.setNodeC sysID_2, sysID_2.NodeD.prop > 10
\end{verbatim}

\texttt{sysID_2.NodeD.prop > 10} is put in WHERE clause of OQL query, and the remaining string
\texttt{id.NodeA.setNodeB sysID_1, sysID_1.setNodeC sysID_2} is added in FROM clause of OQL query.

4. Application in Neuroscience Domain

We have applied our methodology in the neuroscience domain for NeuronBank system [1]. A
conceptual data model (NeuroDM) for neuron in invertebrate animals was designed through the
collaboration of neuroscientists from Georgia State University and us. Figure 4 shows the data
diagram in NeuroDM. The neuron domain-specific functions were proposed and defined by
neuroscientists and implemented by database developers. Based on NeuroDM, the methodology
systematically generates a neuron domain-specific query language (NeuroQL [2]) whose data
sorts and functions are derived from NeuroDM. NeuroQL is an implementation of DSC-QL in neuroscience domain. The mapping between NeuroQL data sorts and database tables/classes is automatically generated by DB-Generator. This mapping will dynamically get updated when the underlying database schema evolves.

Figure 4 Data structure diagram in NeuroDM

Following are three NeuroQL query examples.

**Example 1:** Consider the query: Find all neurons that project to Nerve ‘Pd N 1’ (Note: ‘Pd N 1’ is the name of a nerve). The NeuroQL query expression is:

\[(\text{neuron } n)[\text{project}(n, \text{‘Pd N 1’})]\]

and the corresponding SQL translation is:

\[
\begin{align*}
\text{SELECT } n\text{.name FROM neuron } n, \text{nerve } sysID_1 \\
\text{WHERE sysID}_1\text{.name = ‘Pd N 1’} \\
\text{AND EXISTS (SELECT * FROM Project sysID_2} \\
\text{WHERE sysID}_2\text{.nerve = sysID}_1\text{.nerveID})
\end{align*}
\]

**Example 2:** Consider the query: Find all neurons that have an electrical synapse with neuron ‘R3-13’ (Note: ‘R3-13’ is a neuron name). The NeuroQL query expression is:

\[(\text{neuron } n)[\text{electricalSynapse}(n, \text{‘R3-13’})]\]

and the corresponding SQL translation is:

\[
\begin{align*}
\text{SELECT } n\text{.name FROM neuron } n, \text{neuron } sysID_1 \\
\text{WHERE sysID}_1\text{.name = ‘R3-13’} \\
\text{AND EXISTS (SELECT * from ElectricalSynapse sysID_2} \\
\text{WHERE sysID}_2\text{.preCell = n.neuronID AND sysID}_2\text{.postCell = sysID}_1\text{.neuronID})
\end{align*}
\]
OR \( sysID_2.postCell = sysID_1.neuronID \) AND \( sysID_2.preCell = n.neuronID \)

**Example 3:** Consider the query: Find all neurons that (a) contain molecule ‘5HT’; (b) are presynaptic cell of some chemical synapse; and (c) the soma of the post-synaptic cell of the chemical synapse in condition (b) has a coloration value ‘Orange Circle’. The NeuroQL query expression is:

\[
(neuron n)[\text{Contain}(n, \text{‘5HT’}), \text{chemicalSynapse}(n, s), s.Soma.coloration = \text{‘Orange Circle’}]
\]

and the corresponding SQL translation is:

```sql
SELECT n.name FROM neuron n, molecule sysID_1, neuron s, soma sysID_4
WHERE sysID_1.name = ‘5HT’
AND EXISTS (SELECT * from Contain sysID_2
    WHERE sysID_2.neuron = n.neuronID AND sysID_2.molecule = sysID_1.moleculeID)
AND EXISTS (SELECT * from chemicalSynapse sysID_3
    WHERE sysID_3.preCell = n.neuronID AND sysID_3.postCell = s.neuronID)
AND sysID_4.coloration = ‘Orange Circle’
AND sysID_4.neuron = s.neuronID
```

From the examples above, we can see that the details of databases are hidden from end-users. Support for domain specific functions and dot-path expressions in NeuroQL make it much easier for naïve users with domain knowledge to learn NeuroQL in contrast to traditional query languages such as SQL and OQL. The NeuroQL query statements are much simpler than the corresponding SQL query statements.

### 5. Conclusions

Some researches have been done on the two concepts, conceptual query language and domain-specific query languages. But we have not seen any result to a DBMS-independent conceptual query language with the dynamic support to domain-specific functions. The domain-specific query languages proposed in [11-14] target at a particular domain such as Web Search Engine, Intrusion Analysis, genomic information, and high energy physics analysis, and cannot apply to other domains. ERQL [8] and ConQuer [10] are two conceptual query languages that can apply to any domain. They, however, do not support user-defined domain-specific functions. In addition, ERQL is limited to ER model and relational database system. ConQuer is based on a more powerful model, Object Role Modelling (ORM) [4], but the queries have to be entered in a
pre-developed graphical user interface. As a consequence, it is difficult to integrate ConQuer into an application as a complemental query component. Another approach introduced in [6] is to construct a domain-specific DBMS on top of a persistent object system, ObjectStore [7]. This approach is specifically tailored to the domain of laboratory information system.

The methodology proposed in this paper aims to combine the virtues of domain-specific and conceptual strategies to create domain-specific conceptual data models and query languages that are user-oriented and DBMS-independent, that support domain-specific objects and functions, and that can capture additional domain semantics. It can be applied to any domain. The mapping between the data sorts in DSC-QL and the objects in database enables the query language dynamically to be updated when the underlying database schema evolves. The domain-specific conceptual data model (DSC-DM) can capture additional domain semantics such as meta-attribute, user-defined data types and data structures like set and enumeration, the relationship between an attribute and an entity, and so on. The domain-specific functions can be defined and updated at any time without reprogramming. With the domain knowledge, naïve users can easily master the query language without knowing the details of the underlying database.

From the perspective of application development, although it is better to apply the methodology in the beginning of the development of an application, it is not difficult to integrate it into an existing application system to provide end-users an advanced query tool as a complement to its query capability. In order to integrate the DSC-QL to an existing application, system administrators only need to create a corresponding DSC-DM according to the database schema and define the DSF implementations in DSC-XML. If DSC-QL is used as the middleware layer between the application layer and the database system, it can save huge amount of works to reprogram the application layer when the database evolves.
References

1. NeuronBank: Knowledgebase of Identified Neurons & Synaptic Connections. Website: http://www.neuronbank.org