Towards a Language for Querying OMG MOF-based Repository Systems

Ilia Petrov, Stefan Jablonski

Chair for Database Systems, Department of Computer Science, University of Erlangen-Nürnberg,
Martensstrasse 3, Erlangen, D-91058, Germany
{petrov | jablonski }@informatik.uni-erlangen.de

Abstract. This paper introduces a SQL-aligned declarative query language called mSQL (meta-SQL) for querying OMG-MOF based repository systems. Querying repository systems may be related to querying multi-database systems having a powerful data dictionary. Systems of this kind find extensive application in data-intensive Web applications, information, application and heterogeneous data source integration. Some of the key features of mSQL are: support for higher order queries and schema independent querying, unified handling of repository data and metadata, quantification over repository model elements.

1 INTRODUCTION

Repository systems are "shared databases about engineered artifacts" [2]. They facilitate integration among various tools and applications, and are therefore central to an enterprise. Loosely speaking repository systems resemble data stores with a new and distinguishing feature – a customizable system catalogue. Preserving consistency between the different layers , e.g. custom-defined catalogue, and schema definition and data layer, is a major challenge specific to repositories. In contrast to databases repository systems handle not only application's data but also metadata (both structural and descriptive) and metamodels specifying the structure and the semantics of the data. The multi-layered metadata architecture of a repository system allows for high levels of reflection and model and information discovery.

In this paper we propose a novel declarative querying language for MOF-based repository systems - mSQL (meta SQL). A declarative query language will eliminate the need for programming logic required to retrieve repository objects meeting certain criteria and thus simplify the repository applications. Such access patterns motivated the development of declarative query languages in other data store projects EJBQL [3] or JDOQL. Since mSQL reflects the special characteristics of repository systems it offers a set of advantageous features in comparison to SQL. Due to higher-order the model discovery features in a repository system mSQL queries are independent from the repository data. The goals of mSQL are (a) to provide schema independent querying (as in the case of SchemaSQL) capabilities (partially specified queries); and (b) to introduce a degree of declarative reflection to account to the specifics of the repository systems.

1.1 Architecture of iRM

The use of mSQL in repository systems complements the navigational access to the repository provided by the RMS/API, providing. It helps to eliminate the need of application code to retrieve a certain set of repository objects. Such code is difficult to maintain and evolve. mSQL queries can be used as a repository entry mechanism alternative to the RMS API programmatic access (navigational access). mSQL queries are

significantly simpler to use and maintain, eliminate the need to program the access, the repository may optimize the code and return results faster.

The logical architecture of the iRM module (see Fig. 1) will be briefly described in this section. The iRM/RMS API is the API of the repository management system. It is based on the JMI Reflective API and extends it to handle the M0 data. The iRM/RMS API augments JMI Reflective API with a set of additional functions for the Consistency Manager and notification. The mSQL Wrapper in the iRM/mSQL contains the implementation of the mSQL operators, e.g. selection, projection, Cartesian product or bag constructors. The iRM/mSQL constructs and executes a query execution plan (QEP).

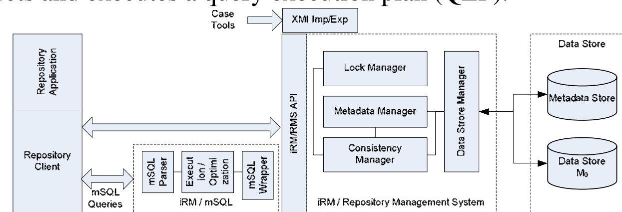

Fig. 1: Architecture of iRM

2 RELATED WORK

There exist multiple technologies for querying metadata. These stem mainly from the fields of federated/mutli- databases and UML-related technologies. In addition there are miscellaneous technologies emanating from the filed of the SemanticWeb (, e.g. R(D)QL) software engineering environments, e.g. PCTE Portable Common Tool Environment) - P-OQL and PNQL.

There are two major of UML-related technologies that may be used for querying- QVT and OCL. The Object Constraint Language [15] is a technology/standard developed by IBM [15] as constraint definition language and query language for model elements. Using OCL expressions a designer/modeler can define complex query expressions which can be used as attribute initialization values, operation implementation, instantiation rules "invariants" for classes and pre and post conditions for operations.

MOF QVT (MOF 2.0 Query, Views, Transformations) [17] is a specification aiming at restructuring and transforming models, generating/deriving other models. QVT is a technology of high importance to the MDA initiative of OMG. The specification is currently in early stages (RFP) there are however some mature proposals [16]. In the context of this paper the query and view aspects are of special relevance. A query is defined as "an expression that is evaluated over a model."[16]. A view is defined as "a model that is completely derived from another model" [16]. Currently there are multiple submission, only a part of which handle the issues of Query and Views. The involved researches and practices point the choice of OCL (with certain extensions) as the underlying technology.

Both of these standards OCL and QVT (queries and views) tend to be used as "embedded" declarations, in a sense that they are meant to be integrated in a model. While mSQL Views can also be used this way, there is a fundamental difference in the way they are used. A repository application utilizes mSQL query very much like a regular database application would utilize database access technologies, e.g. JDBC or ODBC to query a database. In contrast a MOF-based metadata repository supporting OCL would process the expressions internally evaluation them within a repository transaction (in an immediate or deferred manner.)

Another related field is that of multi-database systems; some of the most relevant query languages here are SchemaSQL [4], MSQL [8] and SQL/M [7].

3 DATA MODEL

The data model of mSQL is determined by the JMI Reflective Package and is based on OMG MOF model. The data model comprises the construct of an abstract OO language. The iRM/mSQL data model comprises: Packages (models, schemata), Classes, Attributes, References and Operations, Data types, Associations and AssociationEnds.

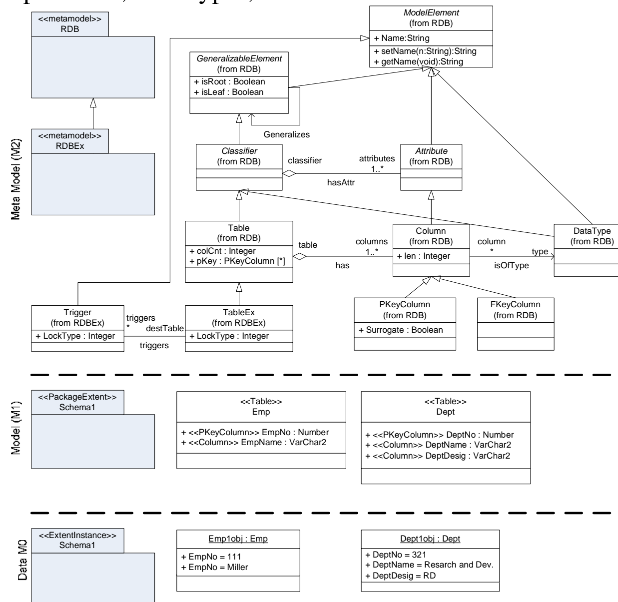

Fig. 2: Example of a Repository Metadta Architectrue

Packages are generic containers for other data model elements such as classes, associations, data types etc. Packages may also relate to other packages: import, specialize, and contain (nested packages). This version of mSQL does not handle nested packages. Handling M0 application (instance) data is yet another issue. We assume that M0 level comprises one M0 data extents, which are instances of M1 package extents (which are, in turn instances of M2 MOF Packages). The metadata standards e.g. MOF or IRDS handle only meta-layers M1 through M3. In the iRM project we implemented M0 support (M0 data extents), which proved advantageous for repository applications, the overall structural integrity of the repository data and querying. The notion of an extent can be extended with respect to all modeling elements – an extent is the set of all instances of a type is called extent. mSQL treats Types (schemata) separately from their extents.

Classes and data types represent types. Classes comprise properties (attributes and references collectively called structural features) and operations. Each class may participate in generalization/specialization hierarchy or in a containment hierarchy. Attributes are of certain data type, which may primitive data type or a complex data type. Complex data types are classes, structures, collections, enumerations. In addition attributes are instances of a class on the next higher level, which we call, attribute type. Attributes have scope (classifier or instance) in mSQL we put special emphasis on the instance-scoped attributes. References are the second kind of properties (for more details see associations). They resemble the relationships in the ODMG model and links in PCTE [9].

Operations represent the behavior of a class. iRM/RMS and hence iRM/mSQL handles operations similar to the ODMG model - only operation signatures are handled, the implementation is not considered. Each operation comprises a set of parameters, each of

which is of certain data type. The return value is a special parameter. Data types can be: primitive or complex.

Associations (relationships) relate model elements in a certain way. Only binary associations are supported. Each association connects two association ends. These ends are of type the respective model elements, e.g. classes or packages, participating in an association. References are attributes, which refer to the opposite association end in an association (i.e. pointers of type the referenced class). Consider, for example, “table” and “columns” Fig. 1. They provide navigational access. Instances of associations are called links.

Type definitions (expressed in terms of model elements) and instances are represented as repository objects. Both are dynamic and can be modified, while the repository management system preserves the consistency automatically. Any repository object has an identity – a unique ID called MofID. This is similar to the OIDs in the ODMG model. Additionally any valid repository object is an instance of exactly one meta-object. Meta-objects represent the types of the instance objects.

We introduce two language literals: Object and Type. Object is the root of the generalization hierarchy of the data model. All instances of various types can be cast to object. All data model elements on a layer (e.g. classes packages) are instances of a Type. Type is subclass of Object. Object is instance of Type. Using Object(x).Type one can retrieve the Type. A type can be cast to object (see Section 3). We will introduce the mSQL syntax in the following section. Fig. 2 presents the set of sample repository data on the different meta-layers. Layer M2 contains two meta-models RDB and RDBEx. Layer M1 contains one model Schema1 instance of M2:RDBEx.

4 mSQL VARIABLES AND QUERIES

All data model elements must be reflected in mSQL syntax, in addition it needs to provide uniform treatment of data and metadata. Therefore mSQL must contain not only variables ranging over the set of instances of a type (as in SQL), but also variables ranging over model elements. This is the case in many higher order query languages e.g. [10]. The syntax of mSQL is inspired and based on one such language – SchemaSQL [4]. The SQL definition of a variable is *<relationName>* *<Var>*, where relationName is the range for the variable Var, ranging over the extent of the relation denoted by relation name. In mSQL in contrast defines seven kinds of variables: (i) meta-lavel variable – identifying the meta-level; (ii) a model (package) variable - identifying the a certain model package on given meta-level; (iii) class variable – ranging over the set of all classes in a package; (iv) structural feature variable – ranging over all attributes and references in a class. Structural feature variables handle also static (classifier scoped) and meta-attributes explained later; (v) data type variable – holding the data type of a structural feature; (vi) association variable – ranging over all associations in a model; and (vii) the normal SQL variable ranging over the set of instances of a given type (or association. This version of mSQL does not support querying operation signatures – therefore no “operation” variables are defined.

Consider for example the following: to identify the M2 Table class (Fig. 2) in mSQL we use the notation M2::RDB:Table. The general format is:

METALEVEL::PACKAGE:CLASS.ATTRIBUTE^DATATYPE, or
METALEVEL::PACKAGE:CLASS~METAATTRIBUTE^DATATYPE

mSQL makes extensive use of nested variables. Several authors [11, 4] state that much of the declarative power of query languages lies in the definition of reference variables in the FROM clause of a query. We use the principle described in SchemaSQL to create nested variable declarations:

<MetaLevel>->M and <MetaLevel>::M->C
<MetaLevel>->M T and <MetaLevel>::M->C P

NIn this way we define M as a variable ranging over all packages or package extents. C is defined as a variable, iterating over all classes in a model M. The variables T and P range over all instances of M and C on the underlying meta-level. For example T stands for all instances of a certain package, while the variable P ranges over the instances of a certain class – in other words ranges over the objects in the class extent.

5 CLASSES, OBJECTS and TYPES

The conceptual model treats classes as schemata for instances called objects. Query Q1 retrieves all classes in all models (outermost packages) on the M2 level.

(Q1) *SELECT P.name, C.name*
FROM M2::->P, M2::P->C;

(Q2) *SELECT P.name, C.name*
FROM M2::->P, M2::P->C
WEHRE ONLY(C) IN M1::Schema1;

P is a variable denoting all models (outermost packages on M2). C is a variable denoting all classes in the respective package P. To traverse the generalization/specialization hierarchy a type participates in, and query classes in that hierarchy mSQL provides three functions: SUPER(), SUB(). As the name implies SUPER() returns all super classes of a class, while SUB() returns the set of all subclasses of a class. Each of these functions has two variants: (a) SUPERm() or SUBm() for selecting only the immediate super or sub-classes; and SUPER() or SUB() return the complete set of super/sub cases. For example SUB(M2::RDB:Table) will return TableEx.

The function ONLY() helps to cope with the querying classes in a generalization hierarchy. It filters only immediate instances of the specified class excluding instances of its subclasses. Q2 is example of a query, which returns the names of all packages and all classes they include, whose direct instances are in the M1::Schema1 package extent

mSQL defines two functions Type() and Object(), in attempt to solve a range of problems: (a) How to handle MofIDs in a generic manner? In OQL or SQL'99 similar problem (relationship, references) is handled by dereferencing (REF, DEREF). In mSQL we have typing, i.e. an object identified by a MofID is always an instance of another one, which makes the issue more complex. Handling identifiers in query languages is discussed in [12]. (b) How to traverse the level hierarchy in a generic manner?

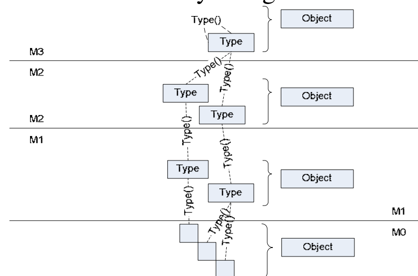

Fig. 3: Type and Object

Every repository artifact can be treated as object (Fig. 3). Every repository object has a type, i.e. it is an instance of a type. Every type can be converted to an object, and be treated as an instance of another type. The converse is not true. Due to self-description M3 types are instances of themselves and can therefore be converted to objects. In order to facilitate handling type inheritance we introduce two variants of the Type function: (i) TYPEm(), returns only the immediate type of an object; (ii) TYPE() immediate type and the set of all of type's (super-classes).

(Q3) *SELECT T.name*
FROM TYPEm(Object(11117)) T

(Q4) *SELECT T.name*
FROM Object(M2::RDB:Table) T

(Q5) *SELECT T.name*
FROM M2::RDB:Table T

Consider queries Q3, through Q5. Q3 returns the name of the immediate type (meta-object) of an artifact/object with MofID 11117 (M1::Schema1:Emp), which is M2::RDB:Table. Q4 returns the same result; "T" in this cases is a single object variable. Q5, however, returns the names of all instances M1 of M2::RDB:Table; "T" in this case is an instance/tuple variable; the result is M1::Schema1:Emp. With Object() and Type() we move across meta –levels; we convert metadata to model-elements. In repository systems this is possible since any repository object has a meta-object, which in turn is also repository object.

6 ATTRIBUTES and ATTRIBUTE TYPES

Every Attribute has an attribute type. An attribute type is not the attribute's data type, which can be primitive or complex. Attribute type is a meta-class, whose instance is the respective attribute definition. SQL'99, in contrast assumes data types. The ability to work with attribute types language is a distinguishing characteristic of mSQL. It increases the level of schema independent querying because mSQL can perform quantification queries on domain specific attribute types. For example select all TableEx tables having an attribute of type PKeyColumn. The concept of attribute types enables mSQL to perform a class of type compatibility queries, which SQL cannot do. Two attributes are compatible not only if their data types are compatible, but also if their attribute types are compatible. In a domain specific environment this helps to reduce the semantic incompatibilities. For example, select all M1 types, in the M1::Schema1 extent, having attributes of attribute type M2::RDB:PKeyColumn.

(Q6) *SELECT C_Inst.Name*
FROM M2::RDB:Class C_Inst, C_Inst ->A
WHERE TYPEm(A) = M2::RDB:PKeyColumn AND C_Inst IN M1::Schema1;

Querying MOF-based meta-models yields a specific problem. How can it be determined that a meta class is the meta-type of attributes (attribute type for M_{n-1} attributes) and not the meta-type for classes. For example how can it be determined that an M2 class Table is the type for classes on M1 and that instances of M2 Column are attributes of M2 Table instances. Distinguishing between these two is critical for executing the definitions C_inst → A or M1::Schema1:Emp1.empno. A in this case is a model element variable ranging over the attributes of all C_Inst classes. The attributes A are not just the values of the attributes of the meta-class, e.g. colCnt, isRoot, isLeaf, name, rather they are instances of attribute types, e.g. M2 Column and its subclasses. In MOF representation, M2 Table instances are represented as complex objects, where the component (part) repository objects are "attributes" i.e. the instances of M2 Column. Therefore executing a query requires explicit designation of "attribute" objects, otherwise queries cannot be performed against a meta-model in the general case.

There are several approaches to problem solution: (i) Proper metamodeling – all attribute types are subclasses of the abstract meta-class Attribute. This approach is considered a proper one from theoretical point of view [13, 14]. (ii) An explicit type casting may be introduced in the query language - Attribute(…). This approach contradicts our goal of schema independent querying because it requires explicit specification. (iii) Extending the MOF Standard. This is undesirable because it will render the whole repository incompatible.

Approach (i) is chosen and implemented in iRM. It requires that any M2 model derives from an abstract M2 model containing a set of abstract classes, e.g., Class, Classifier, Attribute, DataType, Namespace, GeneralizableElement etc., modeling the "abstract

syntax" for the derived meta-model. The abstract meta-model categorizes the model elements of the derived meta-model. For example all subclasses of Attribute are to be considered as attributes of subclasses of Class. Based on the category it can be judged about the "kind" of meta-class, which helps us solve the problem described previously in this section by introducing an artificial "schema". It is a representation of the complex objects, instances of Classes, which can be queried. In this representation attributes and operations of a class are represented explicitly (not as component objects), which avoids the ambiguity when querying. Generated schemata have the form:

```
ClassMetaObject ClassName {
        AttributeMetaObject AttributeName : DataTypeMetaObject DataType […]
        OperationMetaObject Operation : (set of  ParameterMetaObject Parameter
                [role: ReturnValue/Argument]: DataTypeMetaObject DatType
        )
}
```

After generating the schema iRM/mSQL generates a bag of instance objects. The query is then performed against the generated schemata and the set of objects. The general algorithm for constructing the schema is:

```
Repository Object Obj
Schema S
IF GetMetaObject(Obj) ∈ GetSubclass(Class)
        S.ClassName = Obj.Name
        {SComp} = EnumerateComponentObjects( Obj )
        ∀SObj ∈ {SComp}
                IF GetMetaObject( SObj ) ∈ GetSubclass(Attribute)
                        S.AttrName = SObj.Name
                        DTSObj = GetDataType(SObj)
                        S.AttrDataType = DTSObj.Name
                ENDIF;
        ConstructOperations();
ENDIF;
ConstructInstanceBag( S )
END;
```

To select the type of an attribute use the function Type(). It will return immediate Attribute Type and all of its super-types (Q5). Show all M1 types, in the M1::Schema1 extent, having attributes of attribute type M2::RDB:PKeyColumn (Q6)

```
(Q7)SELECT t.MofIF, t.Name
    FROM Type(M1::Schema1:emp1.empno) t;
```

```
(Q8) SELECT   T_Ext.Name
     FROM     M2::RDB->T T_Ext, T_Ext->A
     WHERE    TYPEm(A)=M2::RDB:PKeyColumn
              AND T_Ext IN M1::Schema1;
```

A class may have normal attributes and meta attributes. The difference between them is subtle and results from the meta model. They also play different roles. Normal attributes were already discussed in the previous section. They act as attributes of a class. Their meta-type is a sub-class of M2:Attribute.The meta attributes are values of the attributes of a class. For example the colCnt is a meta attribute for the M1 class Emp and Dept. Any M1 instance class of M2:table must have a value for the meta-attribute colCnt because its multiplicity is 1. To query meta-attributes we use the tilde ("~") character.

```
(Q9) SELECT DISTINCT e.EmpName, e~colCnt, d~colCnt
     FROM M1::Schema1:Emp e, M1::Schema1:Dept d;
```

mSQL offers the ability to query the data types of attributes and test them for compatibility. In this sense two attributes are "compatible" not only if they have the same data type but also if they have the same attribute type. This idea helps, to a certain degree, in avoiding comparisons of semantically different attributes which have same data types. mSQL SIMILAR to test data type compatibility. Compatibility for primitive data types is defined in JMI. Compatibility for complex types is defined in terms of a common super type. As an example consider the following query: select all instances of M2::RDB:Table, which have a primary key attribute of data type VarChar2 (Char and VarChar is compatible with VarChar2).

(Q10)

```
SELECT  T.MofID
FROM    M2::RDB:TableEx T, T->A^D
WHERE   ONLY(TYPEm(A)) = M2::RDB:PKeyColumn AND D.name='VarChar2'
```

7 CONCLUSIONS and FUTURE WORK

In this paper we introduced the mSQL query language. We motivated its use in different areas, such as schema integration, domain driven development, schema independent querying. We introduced the syntax of mSQL and provided examples. mSQL has been implemented in the frame of the iRM project. The current implementation lacks query rewriting capabilities. Introducing algebra and a cost model would contribute to query optimization.

REFERENCES

[1] Petrov, I., Stefan Jablonski, An OMG MOF based Repository System with Querying Capability - the iRM Project. Proceedings of iiWAS. 2004.

[2] Bernstein, P., T. Bergstraesser, J. Carlson, S. Pal, P. Sanders, D. Shutt. Microsoft Repository Version 2 and the Open Information Model. Information Systems 24(2), 1999, pp. 71-98 Conger., S., and K. D. Loch, eds., Ethics and Computer Use, 2nd ed., Elsevier, Amsterdam, p. 105 (1999).

[3] Mnoson-Haefel, Richard. Enterprise Java Beans. Third Edition. O'Reilly. September 2001.

[4] Lakshmanan, Laks V. S., Fereidoon Sadri, Subbu N. Subramanian. SchemaSQL: An extension to SQL for multidatabase interoperability. ACM Transactions on Database Systems (TODS), Volume 26 , Issue 4. December 2001).

[5] Miller R.J. Using Schematically Heteroge neous Structures. ACM SIGMOD, 189-200, Seattle WA. May 1998

[6] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In ACM SIGMOD Int'l Conf. on the Management of Data, pages 393-402, 1992.

[7] Kelley, W., Gala, S. K., Kim, W., Reyes, T.C. and Graham. B. Schema architecture of the UniSQL/M multidatabase system. In Modern Database Systems. 1995

[8] John Grant, Witold Litwin, Nick Roussopoulos, and Timos Sellis. Query languages for relational multidatabases. VLDB Journal, 2(2):153-171, 1993

[9] Wakeman, L., J. Jowett. PCTE - The Standard for Open Repositories. Prentice-Hall. May 1993

[10] Chen W. Kifer M. and Warren D.S. Hilog: A foundation for higher order logic programming. Journal of Logic Porgramming. 15(3) 1993

[11] Yu Clement, Meng Weiyi. Principles of database query processing for advanced applications. Morgan Kaufmnn Inc. 1998

[12] Abiteboul, Serge, Paris Kanellakis. Object Identity as query language primitive. Proceedings of the 1998 ACM SIGMOD International Conference on the Management of Data (Portland, Ore.). ACM, New York, pp. 159 –173.

[13] Clark T., A. Evans, S. Kent: Engineering Modelling Languages: A Precise Meta-Modelling Approach. FASE. pp 159-173. 2002

[14] Ortner, E. Repository Systems Teil 1: Mehrstufigkeit und Entwicklungsumge-bung" and "Repository Systems Teil 2: Aufbau und Betrieb eines Entwicklungs-repositoriums" . Informatik-Spektrum, Volume 22 Issue 4 (1999) and Volume 22 Issue 5 (1999) (in German)

[15] UML 2.0 OCL Specification. OMG document final/03-10-14, November 2003

[16] T. Gardner, C. Griffin, J. Koehler, R. Hauser: A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. MetaModelling for MDA Workshop, York, England, 2003

[17] OMG MOF 2.0 query, views, transformations RFP. OMG document ad/02-04-10, October 2002