Querying Registry-Published Mathematical Web Services

Rebhi Baraka* Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
{rbaraka,schreine}@risc.uni-linz.ac.at

Abstract

This paper describes a light-weight, content-based, functional query language. The Mathematical Services Query Language (MSQL) is specifically designed and implemented for querying mathematical web services described in the Mathematical Services Description Language (MSDL) and published in the MathBroker registry. Based on the client request, MSQL uses the registry querying functionality to retrieve a candidate collection of documents and then uses its own querying functionality to further filter these documents based on their contents.

1. Introduction

Describing, publishing, and discovering web services are crucial issues that have recently received considerable attention. A *mathematical* service is a web service that offers the solution to a mathematical problem (based on e.g. a computer algebra system or on an automated theorem prover). In our MathBroker project [9], the XML-based Mathematical Services Description Language (MSDL) [4] has been developed to adequately describe mathematical services respectively their constituent entities such as problem, algorithm, implementation, and machine. To facilitate the process of publishing and discovering mathematical services, we have developed the MathBroker registry [3] where MSDL descriptions of services are published such that clients can discover them by browsing or querying it.

Figure 1 illustrates the MathBroker information model for the description of mathematical web services. It shows the kinds of entities that can constitute the description of a service and the associations among them.

The entities of the model are:

- **Problem** that can be specified by input parameters, an input condition, output parameters, and an output condition. A problem can be a special version of another problem.
- **Algorithm** that can be described by (a link to the description of) the problem it solves, as well as by time and memory complexity, and termination conditions.
- **Implementation** that can be described by the software used for implementing an algorithm and for the resulting runtime efficiency (absolute efficiency factors for the algorithmic complexity.
- **Realization** that brings together the abstract specification of the service functionality with the actual details of the interface described in the Web Services Description Language (WSDL).
- **Machine** that can be described by its processor type and speed, by its memory size, and by the type of the operating system it uses.

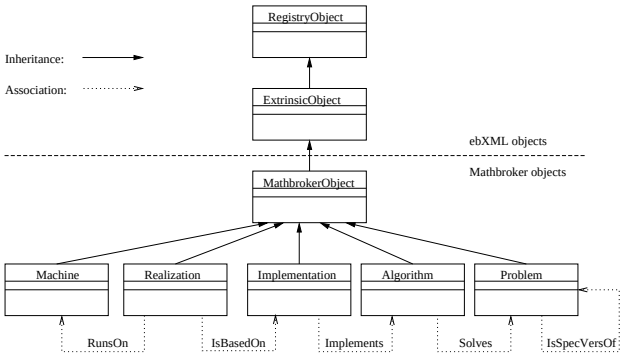

Figure 1. The MathBroker Information Model

A skeleton of a service description in MSDL containing these entities is shown in Figure 2. Mathematical content in this description is written in OpenMath [11] which

* This work was sponsored by the FWF Project P17643-NO4 "Mathbroker II: Brokering of Distributed Mathematical Services".

```
<monet:definitions
  <mathb:machine_hardware name="perseus">
    ...
  </mathb:machine_hardware>
  <monet:problem name="integration">
    ...
  </monet:problem>
  <monet:algorithm name="RischAlg">
    ...
  </monet:algorithm>
  <monet:implementation name="RImpl">
    ...
  </monet:implementation>
  <monet:service name="RRISC">
      ...
  </monet:service>
</monet:definitions>
```

Figure 2. A Skeleton of an MSDL Service Description

is an XML-based format for representing mathematical objects in a semantics-processing way.

The MathBroker registry implementation incorporating the information model provides a set of functionalities for processing, classifying, associating, publishing, and searching MSDL service descriptions in the registry. Service descriptions are classified in the registry according to a predefined classification taxonomy which represents a tree-structured way to classify or categorize published descriptions. A common example of a classification in mathematics is the GAMS (Guide to Available Mathematical Software) classification taxonomy.

Searching facilities of the MathBroker registry are rather limited, they only allow to query metadata accompanying the service when published in the registry. Although metadata such as the name, the unique identifier, the classifications, and the associations of a service are useful in some cases, e.g. when we know the service we want to use, or when we are seeking a service based on its associations to other services, they are insufficient in many cases involving mathematical services as a basis for service discovery, e.g., when we are seeking a service whose problem has a certain input precondition. Therefore it is necessary to resort to the contents of the MSDL description of a service where sufficient and complete information can be found. So to overcome this limitation of the current search facilities of the registry, we have designed and implemented the Mathematical Services Query Language (MSQL) for retrieving and querying the contents of MSDL documents that are published in the MathBroker registry.

MSQL has a set of features compatible with today's XML query languages to make the process of querying, the XML-based, MSDL documents easy and efficient. Our aim is to have a simple, small, yet general querying language that can be extended later to deal with the semantically rich content of MSDL and/or query other XML content. This has lead us to design the language taking into consideration specific characteristics of current XML query languages.

The rest of this paper describes the architecture (Section 2), the features (Section 3), and the implementation (Section 4) of MSQL.

2. The MSQL Architecture

Figure 3 provides a high-level overview of the MSQL architecture which consists of the following parts:

- The *MSQL Engine* which constitutes the querying functionality.
- The *MathBroker Registry* where MSDL documents are published.
- The *Reasoner* which handles the part of a query that needs reasoning (currently being implemented).

A client application sends a query to the MSQL engine and receives a set of MSDL descriptions. These descriptions are either returned to the user or processed further for specific tasks, e.g., to access the service having the resulting description.

The MSQL Engine consists of the following components which are designed according to the query structure explained in Section 3:

- **The Query Processor** which receives the query, divides it into processable parts, and hands each part to its corresponding component.
- **The Registry Handler** which receives from the processor the query part needed to retrieve MSDL documents from the registry based on their types and classifications.
- **The Expression Evaluator** which evaluates the expression part of a query against the documents retrieved from the registry, filters them, and forwards those passing the filter to the Result Quantifier and Sorter component.
- **The Reasoner Interface** which receives from the processor the query part that needs reasoning and sends it to the reasoner.
- **The Result Quantifier and Sorter** which decides whether to return some or every queried document as a result and whether to sort the resulting documents.

In this architecture, an MSQL query received from a client is processed as follows:

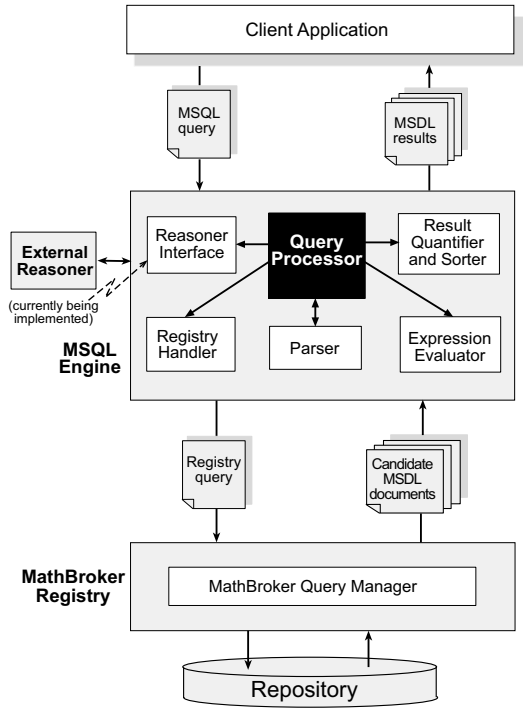

Figure 3. The MSQL Architecture

- The query is parsed according to MSQL grammar and transformed into an abstract syntax tree.
- The engine connects to the registry and invokes the "MathBroker Query Manager" which is part of the registry API. It queries the registry for candidate documents based on the type of document required and based on the classification node under which the required document is classified. It returns to the engine the collection of candidate MSDL documents.
- The Expression Evaluator then evaluates the condition expression of the query against each candidate document and if the document satisfies the condition adds it to the collection of resulting documents.
- The Result Quantifier and Sorter then quantifies and orders the resulting documents depending on kind of query (see the next section).

The engine then returns the resulting MSDL documents to the client application.

3. The MSQL Language

3.1. Requirements

Given a set of MSDL documents (of a structure such as the one in Figure 2) published in the registry, we would like to perform queries of the following kind:

> Find every problem under classification concept "/GAMS/Symbolic Computation" whose first input argument has type "integer".

To accomplish this request, we can do the following:

- consider the registry classification node "/GAMS/Symbolic Computation" and fetch each document with entity type "problem" beneath it;
- process each document and return it, if it satisfies the following criteria: *the first "input" occurring in the "problem" has type "integer"*.

The returned documents may need to be sorted before being returned as a result.

Performing the first step involves contacting the registry and fetching the candidate documents from it. Performing the second step involves processing the returned candidate documents to see if they satisfy the stated criteria. Based on these steps, we defined a set of requirements the query language should meet. They are as follows:

- **Precise semantics:** The language should have a formal semantics to make the intended meaning unambiguous. This semantics should also provide a straight forward reference for evaluating the correctness of the implementation.
- **Functional:** The language should specify what is to be done; how is it done is left to the implementation.
- **Registry interfacing:** The language should have the functionality to access the registry in order to determine and retrieve the candidate collection of MSDL documents.
- **Composing a query:** The language queries should be composed in a concise human-readable query syntax.
- **Query operations:** The operations that have to be supported by MSQL are:
  - **Retrieval:** Retrieving an MSDL document based on the type of entity it describes and on the registry classification concepts under which this entity is classified.
  - **Selection:** Choosing an MSDL document from the candidate documents based on content, structure, or attributes.
  - **Evaluation:** Ultimately selecting a document based on the evaluation of a semantic condition expressed by a logical formula.
- **Input and output:** The input to a query should be MSDL documents satisfying the "Retrieval" requirement. The output of a query are MSDL documents.
- **Result views:** The language should support ordered and unordered views of query results.

It is not required that the language is capable of restructuring a document or returning portions of a document as a result. This is not a disadvantage because our goal is to return whole documents that can be further processed by other applications.

Based on these requirements we have designed the language described in the following subsection.

3.2. The Query Structure

The general structure of a query in MSQL is:

```
SELECT EVERY|SOME <entity>
FROM <classificationConcept>
WHERE <expression>
ORDERBY <expression> ASCENDING|DESCENDING
```

The query has four main clauses: the SELECT clause, the FROM clause, the WHERE clause, and the ORDERBY clause. The FROM clause and part of the SELECT clause, namely `entity`, are registry-oriented, i.e., their functionality is applied to the registry. This satisfies the "Registry interfacing" requirement by determining the type of document, its classification in the registry, and retrieving it from the registry. This is a crucial issue of the language from the point of efficiency role since it limits the range of documents to be queried to those who are of type `entity` and classified under `classificationConcept`. The `entity` types as stated in the information model (see Figure 1) are PROBLEM, ALGORITHM, IMPLEMENTATION, REALIZATION, and MACHINE. `ClassificationConcept` is a node in a given classification taxonomy of the registry, e.g., "/GAMS/Symbolic Computation" in the GAMS classification of mathematical subjects. The SELECT clause also determines whether to return some or all of the resulting documents to the user by its SOME or EVERY quantifier.

The WHERE and the ORDERBY clauses apply their `expression` parts to each candidate document retrieved from the registry. The expression of the WHERE clause is a logical condition: if it is evaluated to true, the document is considered as (part of) the result of the query. The ORDERBY clause sorts the resulting documents in ASCENDING or DESCENDING order based on comparison criteria resulting from the evaluation of its expression on each document.

We illustrate the structure described so far by a concrete query example.

Example 1. *Find all problems under the classification concept "/GAMS/Symbolic Computation" with first input having type integer and order them according to their names in descending order.*

```
SELECT EVERY problem
FROM /GAMS/Symbolic Computation
WHERE //body/input[1]/signature/om:OMOBJ/
       om:OMS[1][(( @name = "Z" )
        and ( @cd = "setname1" ))]
ORDERBY /problem/@name descending
```

This query asks for every "problem", i.e., every document of type "problem" classified under "/GAMS/Symbolic Computation" that satisfies the WHERE expression. The resulting documents are to be sorted in descending order according to their names. The core of the query is its WHERE expression which allows us to express the first input and checks its type (details is given below).

Since we are designing a light-weight query language, we have specified a minimal set of expressions that are necessary to address the contents of the target MSDL documents. MSQL expressions include: path expressions that can access every part of an MSDL document; expressions involving logical, arithmetic, and comparative operators; conditional expressions; quantified expressions; functions; and variable bindings. They satisfy the "Functional" requirement of the language. The formula is applied to an MSDL document modeled as a tree of nodes. A node can be a root, an element node (element), an attribute, or a text. The various kinds of expressions are described below.

Path Expressions These expressions are basically special XPath [6] expressions. A path expression is used to locate nodes within a document tree. It consists of a series of one or more steps, separated by "/" or "//", and optionally starting with "/" or "//".

A "/" at the start of a path expression begins the path at the root of the current node.

A "//" at the start of a path expression begins a sequence that contains the root of the current node plus all nodes descending from it. This node sequence is used as the input to subsequent steps in the path expression. The `//body` step in Example 1 traverses the root node and nodes descending from it until the `body` node which is used as input to the `/input[1]` step.

A step in a path expression generates a sequence of items and then filters the sequence by zero or more predicates. The value of the step consists of those items that satisfy the predicates.

A predicate consists of an expression enclosed in square brackets. The predicate expression can either be logical which evaluates to a truth value or numeric which evaluates to a numeric value. In Example 1, the predicate in the `/input[1]` step is numeric, it specifies the first `input` node.

The `/om:OMS[1][(@name = ''Z'') and (@cd = ''setname1'')]` step has a sequence of two predicates. The first predicate is numeric and it specifies the first `om:OMS` node. The second is logical that uses the `=`

and the `and` operators to evaluate to a truth value. It operates on the `om:OMS` node to check if its `name` attribute has value equal to `Z` and `cd` attribute has value equal to `setname1`. The value of the second predicate represents the value of the expression; if its value is `true` then the current document is selected.

Operators MSQL supports arithmetic, logical, and comparison operators. They are used inside predicates to perform arithmetic, logical, and comparison operations. In Example 1, the expression in the last predicate uses the comparison operator "=" and the logical operator "and" to evaluate to a truth value.

Functions In the context of a predicate, functions may be applied to the current node to extract information used in some operations.

Example 2. *Find all problems in "/GAMS/Arithmetic, error analysis/Integer" that have no precondition.*

```
SELECT EVERY problem
FROM /GAMS/Symbolic Computation
WHERE //body[empty(/pre-condition)]
```

In this example, the `empty` function takes a path expression and returns *true* if the node pointed to by the path is empty. The query uses this function to check if the "body" node has an empty "pre-condition" element node.

Conditional Expressions and Variable Bindings Conditional expressions are used when the document to be returned depends on some condition. An expression or a value that are used in more than one place in the same query can be bound to a variable so it does not need to be defined again.

Example 3. *Find every service in "/GAMS/Linear Algebra" such that if it has an implementation it runs on a machine called perseus or its interface is on the said machine.*

```
SELECT EVERY service
FROM /GAMS/Linear Algebra
WHERE
 if not (/service[empty(//implementation)])
 then
  let $d := doc(//implementation/@href) in
    $d/hardware[contains(@name, "perseus")]
 else //service-interface-description[
        contains(@href, "perseus")]
```

Example 3 shows a query that uses a conditional expression to decide if the current service document node has (`IsBasedOn`) an implementation. If this is the case, it takes the URI of such implementation document and retrieves it from the registry and checks if this implementation is related to the machine `perseus`. If this is not the case, it checks (in the `else`) if the service has its interface on the said machine, i.e., `RunsOn` on the said machine. The `let` clause is used to bind a document to variable `d`. Variable `d` is then used as part of the path expression.

In the same example, the `contains` function returns *true* if its first argument value contains as part of it its second argument value. The `doc` function returns the root node of the document whose name appears as its argument. Its argument is a URI that is used as the address of the required document in the registry.

The query in Example 3 aside from showing the expressiveness of MSQL in dealing with the MSDL content, it also reveals how MSQL utilizes the structure of the MSDL information model (see Figure 1) supported by the registry. It uses associations (e.g., `IsBasedOn`, `RunsOn`) among the entities of the model implicitly in the query.

Quantifiers MSQL provides universal and existential quantifiers to test if every/some element in a document satisfies a certain condition.

Example 4. *Find all problems in "/GAMS/Arithmetic, error analysis/Integer" in which the OpenMath content dictionary "sts" and the "mapsto" symbol are used in the same signature.*

```
SELECT EVERY problem
FROM /GAMS/Arithmetic, error analysis/Integer
WHERE every $p in /problem satisfies
 some $s in $p//signature satisfies
  $s/om:OMOBJ/om:OMA/om:OMS[@cd = "sts"] and
  $s/om:OMOBJ/om:OMA/om:OMS[@name = "mapsto"]
```

The every quantifier requires all "problem" nodes to satisfy the "some" quantifier which checks if at least one signature satisfies the condistion specified.

4. The MSQL Implementation

The implementation of MSQL is based on a formal semantics [1] satisfying the "Precise semantics" requirement. We have used the format of denotational semantics [12] to give formal meaning to each construct defined in the previous section. In the implementation, each mathematical function mapping a syntactic domain to a semantic domain is realized by a Java method whose body corresponds to the mathematical function definition.

In the implementation of the the MSQL engine architecture (see Figure 3) the main functionality is exposed to the user by the MSQL API [2]. The following code outlines one way for using the API in client applications:

```
MsqlQuery msqlQuery = new MsqlQueryImpl();
MathBrokerConnection con =
  msqlQuery.makeConnection(connectionProps);
ChildASTqueryTree =
  msqlQuery.parseQuery(queryString);
Collection resultsCollection =
  msqlQuery.performQuery(queryTree, con);
```

The interface `MsqlQuery` presents the MSQL engine and contains the functionality for accessing the rest of the API. A connection is made to the registry through the `makeConnection` method. A received MSQL query is parsed using the `parseQuery` method. The query is processed and the results are returned all together as a collection using the `performQuery` method.

An alternative to `performQuery` is `iterateQuery` which allows the client to iteratively ask for one document satisfying a query after the other (such that the engine needs not process all candidate documents when the client is satisfied with some result early).

5. Related Work

Existing XML query languages range from ones that support simple node finding and path expressions to more comprehensive ones that support processing, transformation, and querying tasks of XML documents. QUILT [5] is a language that attempts to unify concepts from some of these query languages in order to exploit the full versatility of XML. Its proposal has been adopted latter as the basis for the development of XQuery [7]. XQuery is intended primarily as a query language for querying collections of XML documents or documents viewed as XML (e.g., a SOAP representation of any data source). The language is designed to be broadly capable of dealing with many sources of XML and not only query them, but also process them, and have new XML structures as a result.

Based on our needs, XQuery would be a bulky language that has a lot of functionality that we don't need and does not address some of our requirements such as dealing with classification schemes and types of objects stored in the MathBroker registry. Instead, we used some of its features and of its predecessors in the design of MSQL. MSQL adapts from XQuery [7] the syntax for navigating in the hierarchical structure of MSDL documents. From SQL (Structured Query Language), MSQL utilizes the idea of a series of clauses based on keywords that provide a query pattern (the SELECT-FROM-WHERE pattern in SQL).

Considering mathematical-oriented tools, there is a number of approaches [8] for searching and retrieving mathematical content, they range from basic textual methods to semantic-driven techniques. Their functionality depends on the representation of the target mathematical documents. One approach [10] for achieving service discovery performs matchmaking between representations of tasks (client requests) and capabilities (service descriptions). It is performed only at the level of input, output, pre- and post- conditions which is not sufficient for full service discovery considering the richness of service descriptions.

6. Conclusion

The Mathematical Services Query Language (MSQL) is a light-weight, content-based, functional query language developed for querying mathematical descriptions given in the Mathematical Services Description Language and published in the MathBroker registry. MSQL complements the metadata-based querying facility of the registry.

MSQL is currently syntax-based: queries are only performed on the syntactic structure of documents. We are working to extend MSQL to perform semantic based querying where the engine contacts a reasoner to perform reasoning steps on the underlying semantics of MSDL.

References

[1] Rebhi Baraka, *Mathematical Services Query Language: Design, Formalization, and Implementation*, Tech. report, RISC, Austria, September 2005, See ftp://ftp.risc.uni-linz.ac.at/pub/techreports/.

[2] ———, *Mathematical Services Query Language (MSQL) API*, Research Institute for Symbolic Computation (RISC), September 2005, See http://poseidon.risc.uni-linz.ac.at:8080/results/msql/doc/index.html.

[3] Rebhi Baraka, Olga Caprotti, and Wolfgang Schreiner, *A Web Registry for Publishing and Discovering Mathematical Services*, Proceedings of IEEE Conference on e-Technology, e-Commerce, and e-Service (Hong Kong, March 29 – April 1), IEEE Computer Society, 2005.

[4] Olga Caprotti and Wolfgang Schreiner, *Towards a Mathematical Service Description Language*, International Congress of Mathematical Software 2002 (Bejing, 17–19 August), World Scientific Publishing, 2002.

[5] Don Chamberlin, Jonathan Robie, and Daniela Florescu, *QUILT: An XML Query Language for Heterogeneous Data Sources*, Proceedings of WebDB 2000 Conference.

[6] World Wide Web Consortium, *XML Path Language (XPath) 2.0*, W3C Working Draft, April 2005, See http://www.w3.org/TR/xpath20/.

[7] ———, *XQuery 1.0: An XML Query Language*, W3C Working Draft, April 2005, See http://www.w3.org/TR/xquery/.

[8] I. Dahn and A. Asperti, *Mathematical Knowledge Management and Searchability*, Deliverable D5.4, 2001, See http://monet.nag.co.uk/mkm/MKMNetTN-D5-4.pdf.

[9] *MathBroker – A Framework for Brokering Distributed Mathematical Services*, Research Institute for Symbolic Computation (RISC), September 2005, See http://www/research/parallel/projects/mathbroker2/.

[10] William Naylor and Julian Padget, *Semantic Matching for Mathematical Services*, Proceedings of the Forth International conference on Mathematical Knowledge Management (Bremen, Germany, 15 – 17 July), Springer, 2005.

[11] *The OpenMath Standard*, September 2005, See http://www.openmath.org/cocoon/openmath/index.html.

[12] David A. Schmidt, *Denotational Semantics – A Methodology for Language Development*, Allyn and Bacon, 1986.