Semantic Querying of Mathematical Web Service Descriptions*

Rebhi Baraka and 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 semantic extension to the Mathematical Services Query Language (MSQL). MSQL is a language for querying registry-published mathematical Web service descriptions expressed in the Mathematical Services Description Language (MSDL). The semantic extension allows queries in MSQL to be based on the underlying semantics of service descriptions; the MSQL engine processes these queries with the help of an automated reasoner.

1 Introduction

Semantic-based discovery of Web services is one of the crucial issues that are currently receiving considerable attention in the field of the Semantic Web. In the case of mathematical Web services, this issue is more subtle due to the fact that they operate within semantically rich domains on objects that need proper encoding and specification.

A *mathematical Web 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 the MathBroker project [13], we have developed a framework for mathematical services based on standards such as XML, SOAP, WSDL, and OpenMath. We have developed the XML-based Mathematical Services Description Language (MSDL) [8] to adequately describe mathematical services and their constituent entities. The description of a mathematical service in MSDL may contain information related to the type of the problem, the algorithm(s) used to solve the problem, related problems, machines executing the problem, etc. A skeleton of a service description in MSDL is shown in Figure 1.

To facilitate the process of publishing and discovering mathematical services, we have developed a mathematical registry [3] where MSDL descriptions of services are published such that clients can discover them by browsing or querying the registry. Since the querying facilities of the registry do not support content-based querying, we have developed the content-based Mathematical Services Query Language (MSQL) [1, 5] which is able to perform queries at the syntactical structure of a MSDL service description. However, mathematical objects

* This work was sponsored by the FWF Project P17643-NO4 "MathBroker II: Brokering 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:hardware href=".../perseus"/>
   <monet:algorithm href=".../RischAlg"/>
  </monet:implementation>
  <monet:service name="RRISC">
      ...
   <monet:problem href=".../integration"/>
   <monet:implementation href=".../RImpl"/>
  </monet:service>
</monet:definitions>
```

Fig. 1. A Skeleton of a Service Description

respectively their MSDL descriptions are semantically rich and MSQL does not capture these semantic structures and their relations. This limits the effectiveness of service discovery since it is not based on the semantic information contained in MSDL descriptions. In this paper we present an extension to MSQL that addresses the semantic information contained in service descriptions. This extension adds a number of constructs to the language in order to express predicate logic formulas and adds a semantic evaluator to the MSQL engine to process these formulas with the help of an automated reasoner. The rest of this paper briefly describes the syntactic structure of MSQL (Section 2), the semantic extension to MSQL (Section 3), the MSQL engine architecture and implementation (Section 4), and finally reviews related work (Section 5).

2 The MSQL Syntactic Structure

The Mathematical Services Query Language is a language designed and implemented to query registry-published services based on the contents of their MSDL descriptions. It provides the functionality to interface to a registry and retrieve service descriptions on which queries are performed. Its implementation is based on a formally defined semantics [1].

The Query Structure

A query in MSQL conforms to the following syntax:

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

The query has four main clauses:

- The *SELECT* clause selects EVERY or SOME description of the type specified by *entity* from a given classification scheme in the registry. The *entity* types defined by MSDL are *problem*, *algorithm*, *implementation*, *realization*, and *machine*.
- The *FROM* clause determines the classification scheme from which the specified description is to be selected. Every service respectively its description in the registry is classified according to predefined classification schemes in the registry. The *FROM* clause limits the range of descriptions to be retrieved for querying to those classified under *ClassificationConcept*.
- The *WHERE* clause applies its *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 the comparison criteria resulting from the evaluation of its *expression* on each document.

MSQL is designed such that it has a minimal set of expressions that are sufficient to construct logical statements on the contents of the target MSDL descriptions and that it is able to address the structure of such descriptions. 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. The following is a sample MSQL query that illustrates the usage of some of these expressions.

Example 1. *Find every service in "/GAMS/Symbolic Computation" such that, if it has an implementation, it runs on a machine called "perseus" and otherwise its interface is on this machine.*

```
SELECT EVERY service
FROM /GAMS/Symbolic Computation
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")]
ORDERBY /service/@name descending
```

This query asks for every service description 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 conditional expression (if .. then .. else) is used to decide if the current service document node has an implementation. If this is the case, it takes from the service document the URI of such implementation document, retrieves it from the registry (`let $d := doc(//implementation/@href)`), and checks if this implementation is related to the machine `perseus`. If this is not the case, it checks in the `else` branch, if the service has its interface on the said machine. The `let` clause is used to assign a document to the variable `d` which is then used as part of the path expression. 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 `contains` function returns *true* if its first argument value contains as part of it its second argument value.

Although MSQL provides the functionality to express and perform queries on the syntactic structure of MSDL descriptions, it does not provide the functionality to express and perform queries on their semantic content. In the next section, we present an extension to MSQL that addresses this limitation.

3 A Semantic Extension to MSQL

The Mathematical Services Description Language (MSDL) is capable of representing not only syntactic structures, but also semantic information. This information is expressed in OpenMath [18], an XML-based standard format for representing mathematical objects in a semantics-preserving way. To illustrate this approach, we first present a sample description to show the underlying semantics of MSDL and then show how a query can be constructed that operates on this semantics.

Consider a description of the mathematical problem of indefinite integration (Figure 2). It consists of the following pieces of semantic information:

- Input: $f : \mathbb{R} \to \mathbb{R}$ (lines 3 to 13) which expresses the type $\mathbb{R} \to \mathbb{R}$ of the input and gives it the local name f.
- Output: $i : \mathbb{R} \to \mathbb{R}$ which expresses the type $\mathbb{R} \to \mathbb{R}$ of the output and gives it the local name i.
- Post-condition: $i = \mathit{indefint}(f)$ (lines 17 to 28) which states that the output i equals the indefinite integral of the input f.

The semantic information expressed in this problem description can be used as a basis for discovering suitable services published in the mathematical registry. Suppose a client wants to solve a problem with the following specification:

- Input: a : $\mathbb{R} \to \mathbb{R}$
- Output: b : $\mathbb{R} \to \mathbb{R}$
- Post-condition: $\mathit{diff}(b) = a$ (which states that the differentiated output equals the input).

```
<problem name="indefinite-integration">
 <body>
  <input name="f">
   <signature>
    <OMOBJ>
     <OMA>
      <OMS cd="sts" name="mapsto"/>
      <OMS cd="setname1" name="R"/>
      <OMS cd="setname1" name="R"/>
     </OMA>
    </OMOBJ>
   </signature>
  </input>
  <output name="i">
    ...
  </output>
  <post-condition>
   <OMOBJ>
    <OMA>
     <OMS cd="relation1" name="eq"/>
     <OMV name="i"/>
     <OMA>
      <OMS cd="calculus1" name="indefint"/>
      <OMV name="f"/>
     </OMA>
    </OMA>
   </OMOBJ>
  </post-condition>
 </body>
</problem>
```

Fig. 2. An MSDL Problem Description

The client would thus like to find some service which solves a problem p such that

$$\begin{aligned} type(input_p) = \mathbb{R} \rightarrow \mathbb{R}\ \wedge \quad (1) \\ type(output_p) = \mathbb{R} \rightarrow \mathbb{R}\ \wedge \quad (2) \\ \forall\, a \in \mathbb{R} \rightarrow \mathbb{R},\, b \in \mathbb{R} \rightarrow \mathbb{R}\ (post_p(a,b) \Rightarrow \mathit{diff}\,(b) = a) \quad (3) \end{aligned}$$

where formulas (1) and (2) state that the types of the input and output shall be $\mathbb{R} \rightarrow \mathbb{R}$ and the universally quantified subformula (3) states that the post-condition $post_p$ of the problem p implies that the differentiation of the output b equals the input a. The truth of this statement depends on knowledge available about the operation *diff*, e.g. a knowledge base may contain the formula $\mathit{diff}\,(\mathit{indefint}(a)) = a$ which semantically relates the operators *diff* and *indefint*.

To express such a formula in MSQL, we extended the grammar of MSQL as shown in Figure 3 by adding two clauses:

```
      <msqlQuery> ::= 'SELECT' ( 'EVERY' | 'SOME' ) <entity>
                      ( 'FROM' <classification> )?
                      ( 'WHERE' <msqlExpr> )?
                      ( 'ORDERBY' <msqlExpr )?;
                    ...
       <msqlExpr> ::= ... | <typematch> | <semanticExpr>;
      <typematch> ::= 'typematch' (omObjExpr, omObjExpr);
   <semanticExpr> ::= 'satisfy' ( <omObjExpr> );
      <omObjExpr> ::= <omApplication> | <omAttribution> | <omBinding>
                      | <omInt> | <omVar> | <omString> | <omSymbol>
                      | <var>;
  <omApplication> ::= 'oma' '(' <omObjExpr> (, <omObjExpr> )* (
                       <varReplacement> )? ')';
  <omAttribution> ::= 'omattr' '(' <omObjExpr>, ( <omObjExpr>
                      <omObjExpr> )(, ( <omObjExpr> <omObjExpr> ))*
                      ( <varReplacement> )? ')' ;
      <omBinding> ::= 'ombind' '(' <omObjExpr> '[' <omBoundVariable>
                        (, omBoundVariable )* ']' <omObjExpr>
                        ( <varReplacement> )? ')';
<omBoundVariable> ::= 'omvar' ':' ( <var> | <omVar> ) '@' '('<omObjExpr>,
                       <omObjExpr> ( <varReplacement> )? ')';
 <varReplacement> ::= '[' <omObjExpr> '/' <var> (, <omObjExpr> '/'
                       <var> )* ']';
          <omInt> ::= 'omi' ':' <number>;
          <omVar> ::= 'omv' ':' ( <letter> | <var> );
       <omString> ::= 'omstr' ':' <letter> ;
       <omSymbol> ::= 'oms' ':' <letter> ':' <letter>;
            <var> ::= '$' <letter>;
                    ...
```

Fig. 3. The MSQL Semantic Extension Grammar

- The clause 'typematch(a,b)' states that type a matches (i.e. equals or is a special version of) type b.
- The clause 'satisfy e' states that the semantic interpretation of the predicate logic formula e (encoded as an OpenMath expression) yields *true*.

The `<semanticExpr>` rule and its subrules define the grammar of predicate logic formulas based on the classification of OpenMath objects into *basic* objects and *compound* objects [18]. *Basic* objects include *Integers*, *Strings*, *Variables*, and *Symbols*. *Compound* objects include *Application*, *Attribution*, and *Binding*. The syntax is defined such that expressions are written in a prefix notation which is internally transformed to OpenMath syntax. For instance the `<omBinding>` subrule (see also Example 2) expresses an OpenMath *Binding* object which is constructed from an OpenMath object (the binder), and from zero or more variables (the bound variables) followed by another OpenMath object (the body). The MSQL expression

```
oma(oms:relation1:eq, oma(oms:calculus1:diff, omv:b), omv:a)
```

is thus transformed to the OpenMath XML object

```
<OMA>
 <OMS name="eq" cd="relation1"/>
  <OMA>
    <OMV name="diff" cd="calculus1"/>
    <OMV name="b"/>
  </OMA>
 <OMV name="a"/>
</OMA>
```

Example 2. Our request to *find some service with problem* p *such that the type checks (1) and (2) and the subformula (3) are satisfied* can be expressed by the following MSQL query:

```
SELECT SOME service
FROM /GAMS/Symbolic Computation
WHERE let $p:= doc(//problem/@href) in
          $a:= $p//input/@name,
          $b:= $p//output/@name,
         $ta:= $p//input/signature/OMOBJ,
         $tb:= $p//output/signature/OMOBJ,
       $post:= $p//post-condition/OMOBJ in
 (typematch(oma(oms:sts:mapsto(oms:setname1:R,
                      oms:setname1:R)), $ta)) and
 (typematch($tb, oma(oms:sts:mapsto(oms:setname1:R,
                           oms:setname1:R)))) and
 (satisfy(ombind(oms:quant1:forall
  [omvar:$a@(oms:sts:type, $ta),
   omvar:$b@(oms:sts:type, $tb)]
  oma(oms:logic1:implies, $post,
      oma(oms:relation1:eq,
          oma(oms:calculus1:diff, omv:$b), omv:$a)))))
```

Variable `$p` represents the problem description of the service retrieved from the registry by the *doc* function according to the problem *href* provided as part of the service description. Variables `$a` and `$b` represent the names of the input and the output of the problem. Variables `$ta` and `$tb` represent the types of the input and the output of the problem. Variable `$post` represents the post-condition of the problem.

The two `typematch` expressions correspond to formulas (1) and (2). They check if type $\mathbb{R} \rightarrow \mathbb{R}$ matches the type `$ta` of the input and if the type `$tb` of the output matches type $\mathbb{R} \rightarrow \mathbb{R}$.

The `satisfy` expression corresponds to the universally quantified subformula (3).

In the next section, we explain how the query is handled by the query engine.

4 The MSQL Architecture And Implementation

MSQL including its semantic extension has been implemented as a query engine [1, 5] and has been incorporated into the MathBroker framework [13] for service publication and discovery.

Architecture

Figure 4 illustrates the architecture of the MSQL engine which consists of the following components:

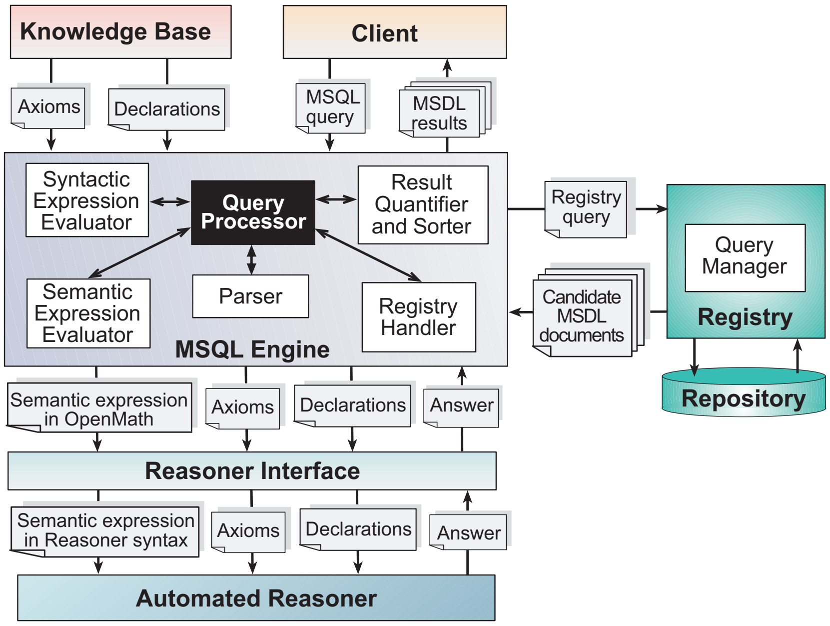

Fig. 4. The MSQL Engine Architecture

- The MSQL **Query Engine** which has the MSQL query functionality. It consists of the following components:
  - The **Query Processor** which receives the query from the client, decomposes it into processable parts, and hands each part to the corresponding component.
  - The **Parser** receives the query from the processor and parses it according to the MSQL syntax. If the query does not comply with the syntax, an error message is returned to the processor which forwards the message to the client.

- The **Registry Handler** receives from the processor the *entity* and *classificationConcept* parts of the query. It composes a registry query to retrieve *EVERY/SOME* description document of the given *entity* type classified under the given *classificationConcept*.
- The **Syntactic Expression Evaluator** receives from the Query Processor the syntactic expression part of the query and evaluates it against each description document retrieved from the registry. It returns to Query Processor those documents for which the expression evaluates to *true*.
- The **Semantic Expression Evaluator** receives from the Query Processor the semantic expression part of the query and evaluates it against each description document retrieved from the registry. It returns to the Query Processor those documents for which the expression evaluates to *true*. Unlike the Syntactic Expression Evaluator, the Semantic Expression Evaluator does not perform the whole evaluation by itself. It rather takes the semantic expression, converts it into OpenMath format (see Figure 5), retrieves from the Knowledge Base the axiom(s) and type declaration(s) needed to reason about the semantic expression and sends all of them to the Reasoner Interface. As required by the Reasoner Interface, the axioms are represented in OpenMath format and the declarations are represented in OMDoc [17] format.
- The **Result Quantifier and Sorter** receives from the Query Processor SOME/EVERY document filtered by the two evaluators, orders (if needed) the documents according to the ORDERBY expression, and returns them as the query result to the Client.

– The **Registry** which stores a collection of published MSDL documents of different *entity* types and classifies them according to some registry-predefined classification schemes. Query requests to the registry are handled by the Query Manager of the registry.

– The **Reasoner Interface** receives from the Semantic Expression Evaluator the semantic expression part of the query in OpenMath, the axiom(s) in OpenMath, and the declaration(s) in OMDoc and converts each one to the format required by the Automated Reasoner and hands them to the reasoner. It gets the answer from the reasoner and sends it to the Semantic Expression Evaluator. The Reasoner Interface used is a component of the RISC ProofNavigator [20].

– The **Automated Reasoner** reasons about semantic expressions based on the axiom(s) and declaration(s) given and returns the answer to the Reasoner Interface. The Automated Reasoner currently used is the Cooperating Validity Checker Lite (CVCL) [6].

– The **Knowledge Base** holds declarations of OpenMath symbols that may be used in semantic queries together with axioms that describe the semantics of that symbols.

Performing the Semantic Query

Based on this architecture, we summarize the actions taken to perform the query in Example 2:

- The Query Engine receives the query from the Client and hands it to the Query Processor
- The Query Processor asks the Parser to parse the query according to the MSQL syntax. If the query does not comply with the MSQL syntax, an error message is returned to the user.
- The Query Processor decomposes the query to processable parts. It hands the registry-related part (the *entity service* and the *classificationConcept "/GAMS/Symbolic Computation"*) to the registry handler.
- The Registry Handler forms a registry query based on the *entity* and the *classificationConcept*, connects to the Registry and hands the registry query to the Query Manager of the Registry which performs the query and returns a set of candidate *service* documents to the Registry Handler.
- The Query Processor asks the Syntactic Expression Evaluator to evaluate the syntactic expression part on the current *service* document. The Syntactic Expression part consists of a *let* expression which has six assignment subexpressions. The evaluations of these subexpressions assign values to variables $a, $b, $ta, $tb, and $post representing input, output, input type, output type, and post-condition respectively. These variables are used in the semantic expression of the query.
- The Query Processor asks the Semantic Expression Evaluator to evaluate the semantics expression against the (same) current *service* document. The Semantic Evaluator performs the following steps:
  - It performs the type checking required by the two typematch expressions. If the result of the check is *true* it proceeds to the next step. Otherwise it returns *false* and the Query Processor proceeds to perform the query on the next candidate document.
  - It converts the `satisfy` expression to OpenMath format. The OpenMath representation of the satisfy expression is shown in Figure 5. The conversion also takes care of variable substitution (e.g., variable `$a` is substituted by the input name f).
  - It retrieves from the Knowledge Base the declarations of the symbols *diff* and *indefint* represented in OMDoc. The two symbols occur in the OpenMath representation of the `satisfy` formula after variable substitution. The declaration of the *diff* symbol is shown in Figure 6. The *indefint* symbol has a similar declaration.
  - It retrieves from the Knowledge Base the axiom $\mathit{diff}(\mathit{indefint}(a)) = a$. This axiom is represented by the following quantified formula in OpenMath format (similar to the OpenMath format of the *satisfy* expression)

$$\forall f \in \mathbb{R} \to \mathbb{R}\ (\mathit{indefint}(\mathit{diff}(f)) = f)$$

```
1 <OMOBJ>
2  <OMBIND>
3   <OMS name="forall" cd="quant1"/>
4   <OMBVAR>
5    <OMATTR>
6      <OMATP>
7        <OMS name="type" cd="sts"/>
8        <OMA>
9         <OMS name="mapsto" cd="sts"/>
10        <OMS name="R" cd="setname1"/>
11        <OMS name="R" cd="setname1"/>
12       </OMA>
13     </OMATP>
14     <OMV name="f"/>
15    </OMATTR>
16    <OMATTR>
17     ...
18     <OMV name="i"/>
19    </OMATTR>
20  </OMBVAR>
21  <OMA>
22   <OMS name="implies" cd="logic1"/>
23   <OMA>
24    <OMS name="eq" cd="relation1"/>
25    <OMV name="i"/>
26    <OMA>
27     <OMV name="indefint" cd="calculus1"/>
28     <OMV name="f"/>
29    </OMA>
30   </OMA>
31   <OMA>
32    <OMS name="eq" cd="relation1"/>
33    <OMA>
34     <OMV name="diff" cd="calculus1"/>
35     <OMV name="i"/>
36    </OMA>
37    <OMV name="f"/>
38   </OMA>
39  </OMA>
40 </OMBIND>
41 </OMOBJ>
```

Lines 5 to 15 represent the conversion of the binder expression

```
omvar:$a@(oms:sts:type, $ta)
```

with the variables $a and $b substituted by their values. It represents the declaration

$$f : \mathbb{R} \to \mathbb{R}$$

Lines 21 to 39 represent the conversion of the *satisfy* subexpression

```
oma(oms:logic1:implies, $post,
    oma(oms:relation1:eq,
         oma(oms:calculus1:diff,
            omv:$b), omv:$a))
```

with the variables $post, $a, and $b appropriately substituted by their values. It represents the implication

$$i = \mathit{indefint}(f) \Rightarrow \mathit{diff}(i) = f.$$

Fig. 5. OpenMath Representation of the *satisfy* Expression in Example 2

- It hands the satisfy expression (in OpenMath), the declarations (in OMDoc), and the axiom (in OpenMath) to the Reasoner Interface which converts each of them to the syntax required by the reasoner. The reasoner decides about the truth value of the expression based on the given axiom and declarations and returns the answer to the RISC ProofNavigator which in turn returns the answer to the Semantic Expression Evaluator.

– If the evaluation of the semantic expression yields *true*, the Query Processor returns the current *service* document to the Result Quantifier and Sorter which returns it to the Client as the ultimate result (because of the SOME clause) of the query. If the evaluation is *false* the Query Processor proceeds to process the query on the next candidate *service* document. If the evaluation is *false* for all candidate documents, then no document is returned as a result of the query.

```
<omdoc:omgroup>
 <omdoc:symbol kind="object" name="calculus1_diff">
  <omdoc:type system="simply_typed"
   xml:id="calculus1_diff_type">
   <om:OMA>
     <om:OMS cd="sts" name="mapsto"/>
     <om:OMA>
       <om:OMS cd="sts" name="mapsto"/>
       <om:OMS cd="setname1" name="R"/>
       <om:OMS cd="setname1" name="R"/>
     </om:OMA>
     <om:OMA>
       <om:OMS cd="sts" name="mapsto"/>
       <om:OMS cd="setname1" name="R"/>
       <om:OMS cd="setname1" name="R"/>
     </om:OMA>
    </om:OMA>
  </omdoc:type>
 </omdoc:symbol>
</omdoc:omgroup>
```

Fig. 6. The *diff* variable declaration

A Prototype Implementation

A prototype of the architecture has been implemented in Java making use of the ebXML registry standard [10] as a basis for the Registry [4], a component of the RISC ProofNavigator [20] as the Reasoner Interface, and the Cooperating Validity Checker Lite (CVCL) [6] as the Automated Reasoner.

The implementation of the MSQL engine is based on a formal definition [1] using denotational semantics [21]. The implementation consists of a set of evaluation classes with a set of methods each of which corresponds to one equation in the denotational semantics. The signature of a method corresponds to the signature of the semantic function. For example, the equation

$$\mathbf{E}[\![V]\!]\ d\ n\ r = lookup(d,\ [\![V]\!])$$

with the semantic function

$$\mathbf{E} : Expression \times Declaration \times Node \times Registry \rightarrow Value$$

is implemented by the Java method

```
static private Value evaluateVariableExpr(ChildAST expr,
                      Declaration declaration,
                      Node node) throws MsqlException {
  VarExpr varExpr = (VarExpr)expr;
  return declaration.lookup(varExpr);
}
```

The prototype implementation of the MSQL engine including its API can be found in [2].

5 Related work

Service discovery is a crucial phase in the service life cycle. Apart from the MathBroker project, two other projects have focused on the description and discovery of mathematical Web services.

The MONET [14] project defines a set of ontologies to model service descriptions (which are basically ontological conversions of MSDL [9, 15] descriptions) as well as queries on those descriptions. These ontologies are written in OWL [19] and are used by a component within the MONET architecture called *Instance Store* [12]. *Instance Store* uses the Description Logic reasoner RACER [11] for matching queries to appropriate services.

MONET uses two classes of ontologies: those describe models internal to MONET (e.g., problem and software ontologies) and those describe models external to MONET (e.g., OpenMath and GAMS ontologies). Individual ontologies of both classes are imported into one MONET ontology. When a service is submitted to the MONET broker, its description is presented in MSDL. This description is transformed to the OWL Abstract Syntax [7] by means of an XSLT stylesheet. Service matching is then performed by submitting a query to the *Instance Store* in the form of an OWL description. The *Instance Store* answers the query by using a combination of Description Logic reasoning and database queries. The reasoning process in the case of MONET is based on a restricted form of first order logic which is more tractable for automated reasoning but strictly less expressive. In our semantic queries, we use full predicate logic which is a highly expressive language.

A matching-based discovery approach [16] to registry-published mathematical services performs matchmaking between representations of tasks (client requests) and capabilities (service descriptions). The approach applies a normalization process on a task. It then compares the normalized task with a registered capability calculating a similarity value that is used in the matchmaking process. Task normalization amounts to carrying out a sequence of transformations on the task description rewriting all logical parts in disjunctive normal form, flattening arguments of n-associative operations, and consistent variable renaming.

The similarity value is calculated based on the matching of the capability precondition (or the task postcondition) and the capability postcondition (or the task precondition). Matchmaking is performed by: registering capabilities in the database, taking a description of a task normalizes it, and returns an ordered list of the capabilities from the registry database based on their calculated similarity.

The matching process used in the discovery is ultimately based on the syntactic similarity traced between tasks and capabilities. In our case, the decision is based on logical implications between statements extracted from descriptions, which is strictly more general.

6 Conclusion

The Mathematical Services Query Language (MSQL) is a language developed for querying mathematical descriptions given in the Mathematical Services Description Language (MSDL) and published in the MathBroker registry. MSQL supports syntax-based queries on the syntactic structure of mathematical service specifications. The semantic extension of MSQL enables it to support semantic-based queries on the underlying semantical structures of such mathematical service specifications. The query engine of MSQL performs semantic-based queries with the help of an automated reasoner which takes predicate logic formulas, decides their validity, and returns the answer to the engine.

A future extension to the presented framework may involve service compositions: when a client submits a service request, a broker agent determines suitable service compositions satisfying the client request and returns the description of a composition rather than that of a single service. To find the suitable candidate services, the agent might form MSQL queries based on information contained in the client request, send them to the query engine, and make composition decisions based on the results returned by the query engine.

References

1. Rebhi Baraka. Mathematical Services Query Language: Design, Formalization, and Implementation. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2005. See ftp://ftp.risc.uni-linz.ac.at/pub/techreports/.
2. Rebhi Baraka. Mathematical Services Query Language (MSQL) API. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 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. In *Proceedings of IEEE Conference on e-Technology, e-Commerce, and e-Service*, Hong Kong Baptist University, Hong Kong, March 29 – April 1, 2005. IEEE Computer Society, Los Alamitos, CA.
4. Rebhi Baraka, Olga Caprotti, and Wolfgang Schreiner. A Web Registry for Publishing and Discovering Mathematical Services. In *EEE*, pages 190–193. IEEE Computer Society, 2005.
5. Rebhi Baraka and Wolfgang Schreiner. Querying Registry-Published Mathematical Web Services. In *Proceedings of The IEEE 20th International Conference on Advanced Information Networking and Applications (AINA 2006)*, Vienna, Austria April 18 – April 20, 2006. IEEE Computer Society.
6. Clark W. Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker Category B. In *Proceedings of 16th International Conference on Computer Aided Verification*, Boston, MA, USA, July 13-17, 2004. Springer.
7. Sean Bechhofer, Peter F. Patel-Schneider, and Daniele Turi. OWL Web Ontology Language Concrete Abstract Syntax. Technical report, The University of Manchester, UK, December 2003. See http://owl.man.ac.uk/2003/concrete/latest/.

8. Olga Caprotti and Wolfgang Schreiner. Towards a Mathematical Service Description Language. In *International Congress of Mathematical Software ICMS 2002*, Bejing, China, August 17–19, 2002. World Scientific Publishing, Singapore.
9. Mike Dewar, David Carlisle, and Olga Caprotti. Description Schemes for Mathematical Web Services. In *EuroWeb 2002: The Web and the Grid: From e-Science to e-Business*, Oxford, UK, December 2002. British Computer Society Electronic Workshops in Computing.
10. ebXML Registry Services Specification v2.0. OASIS, April 2002. http://www.oasis-open.org/committees/regrep/documents/2.0/specs/ebRS.pdf.
11. Volker Haarslev and Ralf Moller. Description of the RACER system and its applications. In *Automated reasoning: First International Joint Conference, IJCAR 2001*, Siena, Itally, June 18–23, 2001. volume 2083 of Lecture Notes in Artificial Intelligence, New York, NY, USA, 2001. Springer Verlag Inc.
12. Instance Store - Database Support for Reasoning over Individuals. The University of Manchester, 2002. See http://instancestore.man.ac.uk/instancestore.pdf.
13. MathBroker II: Brokering Distributed Mathematical Services. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, April 2006. See http://www.risc.uni-linz.ac.at/research/parallel/projects/mathbroker2/.
14. MONET — Mathematics on the Web. The MONET Consortium, April 2004. http://monet.nag.co.uk.
15. Mathematical Services Description Language (MSDL). Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, April 2004. http://poseidon.risc.uni-linz.ac.at:8080/mathbroker/results/xsd.html.
16. William Naylor and Julian Padget. Semantic Matching for Mathematical Services. In *Proceedings of the Forth International conference on Mathematical Knowledge Management*, Bremen, Germany, 15 – 17 July, 2005. Springer.
17. OMDoc: A Standard for Open Mathematical Documents. MathWeb.org, September 2005. See http://www.mathweb.org/omdoc/.
18. The OpenMath Standard. The OpenMath Society, April 2006. See http://www.openmath.org/cocoon/openmath/index.html.
19. OWL Web Ontology Language Reference. W3C, February 2004. See http://www.w3.org/TR/2004/REC-owl-ref-20040210/.
20. The RISC ProofNavigator. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, March 2006. See http://www.risc.uni-linz.ac.at/research/formal/software/ProofNavigator/.
21. David A. Schmidt. Denotational Semantics – A Methodology for Language Development. Allyn and Bacon, Boston, 1986.