Semantic Matching for Mathematical Services

William Naylor and Julian Padget

Department of Computer Science, University of Bath, UK
{wn,jap}@cs.bath.ac.uk

Abstract. The amount of machine oriented data on the web is increasing, simultaneously deployment of agent/Web Services is increasing. This poses a service-discovery problem for client agents wishing to discover Web Services to perform tasks. We discuss a prototype mathematical service broker and look at an approach to circumventing the ambiguities arising from alternative but equivalent mathematical representations occurring in mathematical descriptions of tasks and capabilities.

1 Introduction

The amount of machine-oriented data on the Web is increasing rapidly as semantic Web technologies achieve greater up-take. At the same time, the deployment of agent/Web Services is increasing and together create a problem for software agents that is the analog of the human user searching for the right HTML page. We describe a broker architecture (displayed in figure 1), which is specifically targeted at the discovery of mathematical services.

If a service is to be found and used, it must advertise itself. There are many generic aspects of a service and several specific to mathematical services, but for the purposes of this paper, we will concentrate on four, *viz.* the signatures of inputs and outputs and the pre- and post-conditions which specify the service's requirements and capabilities.

The role of the broker is to act as an intelligent mediator between clients and services, selecting services that match the client's problem statement (task). This task description must specify the signatures of inputs, outputs and the pre- and post-conditions that characterise the service that is sought. The broker's job is to identify service(s) or combinations of services satisfying the attributes given in the task. One of the major problems we address here is dealing with the alternative but equivalent representations that occur in mathematical descriptions of task and capability.

2 Encodings for Mathematical Services

The MONET [MON] project was a European Union (EU) project aiming at bringing together users and providers of mathematical services over the internet. Various ontologies were defined as part of the project which are utilised in our brokerage mechanism. Schemas using these ontologies include the following:

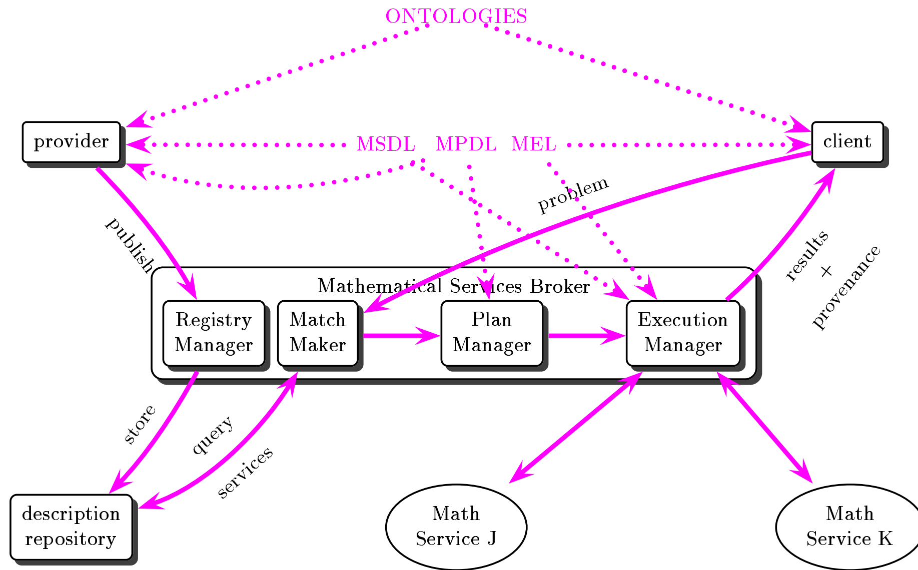

Fig. 1. MONET – Brokerage Mechanism

- Mathematical Service Description Language (MSDL) see [MSL]. This schema was developed to allow the service description mentioned in section 1. It is this language which we shall be mostly concerned with in this paper, as it allows representation of all of the concepts we require for service discovery.
- Mathematical Problem Description Language (MPDL) see [MPL]. This schema allows a client to pose a mathematical problem by specifying an MSDL element, which embodies the problem, as part of the `output` or `post-condition` element, perhaps along with a directive specifying what to do with the problem (evaluate, lookup, prove, etc.). A service may then receive this message and perform the problem posed.
- Mathematical Explanation Language (MEL) see [MEL]. With this schema the service is able to return the results of a calculation to a client. Facilities also exist for returning an explanation of the results, this may be a trace of execution, the proof of a result, supporting evidence, a reference to a formulae evaluated etc. A required element in this ontology is a reference to identify which problem this is replying to.

Naturally, at various points in the above documents, it is necessary to be able to refer to mathematical objects. The schemas are agnostic on how these objects

should be described or what ontologies should be used. In practice there are two main contenders: i) Content MathML [MML] and ii) OpenMath [OM]. MathML has the advantage that there there is greater awareness, and hence more tools have been written which understand it, also there is the close association with Presentation MathML which makes the rendering of objects trivial. OpenMath has the advantage that it is an extensible language, whereas MathML only covers a fixed dialogue of concepts and requires an external definition mechanisms to define other concepts indeed OpenMath is suggested as the definition mechanism in the MathML specification.

3 Normal Form

There are many equivalent ways to describe mathematical conditions: for example if $i, j \in \mathbb{Z}$, then $i \leq j$ and $i - 1 < j$ are equivalent. This implies that the descriptions of mathematical services may not be unique, and consequently creates a significant problem for the broker in identifying mathematical services, as the descriptions must necessarily often contain complex mathematical objects. This is the problem we currently face and we will now describe how we are tackling it. We have observed that most expressions take the form of Q(L(R)) where:

- Q is a quantifier block e.g. $\forall x \exists y$ s.t. $\cdots$
- L is a block of logical connectives e.g. $\wedge, \vee, \Rightarrow, \cdots$
- R is a block of relations. e.g. $=, \leq, \geq, \neq, \in \cdots$ and

The block Q along with the input and output elements serve to define the scope of variables within the document. This will be relevant for steps 4 and 5 of our method. The block L consists of logical connectives which are relevant to step 1. The block R consists of relations which will restrict the value of one or more variables. Examples of relations are equalities or inequalities between expressions, or boolean valued set operators $(\subset, \subseteq, \in, \cdots)$.

Despite the fact there can be no absolute normal form [DSR], we can nevertheless carry out a sequence of transformations to construct a normal form suited to our brokerage task, thus:

1. The logical parts of the task and capability are rewritten in Disjunctive Normal Form (DNF) (see for example [DNF]), which is convenient for the calculation of similarity values (see §5). It also proves convenient for deducing dependencies allowing service composition mentioned in section 7. The transformations use a number of basic re-write rules, for example de-morgens rule, distributivity of $\wedge$ over $\vee$, etc.

   Example 1.
   $(a \wedge b) \Rightarrow \neg(c \vee \neg d)$ rewrites to the equivalent expression $\neg a \vee \neg b \vee (\neg c \wedge d)$ which is in DNF

2. Associativity. Various operations are n-associative e.g. $+, *, \cup, \cap, \wedge, \vee$, this means the operation takes n arguments and if the operation is denoted by $\otimes$, then: $a \otimes (b \otimes c) = (a \otimes b) \otimes c$. A natural form is to *flatten* the arguments,

i.e. provide each argument as an argument of the operation at the first level. i.e.

$$a \otimes (b \otimes c) \to \otimes(a, b, c) \text{ and } (a \otimes b) \otimes c \to \otimes(a, b, c)$$

3. The numerous domain specific mathematical equivalences that exist, eg. the example mentioned in the first paragraph of section 3, are addressed by means of a database of context sensitive rules.
   We use a number of techniques for deducing context information:
   i) Performing a prior pass of the document and looking for constructs such as $x \in \mathbb{Z} \Rightarrow \cdots$.
   ii) Scanning the sts (Small Type System [STS]) files which record information about the signatures of OpenMath symbols.
   iii) Scanning signatures from the input and output variables given in the MSDL.

   An important consideration is that cycles must not exists whilst applying these rules i.e. we must not have the two rules $A \to B$ and $B \to A$ concurrently in our system (or any generalisation of this case).
4. α' conversion[1] addresses the situation where there are bound variables in the description as a result of quantification or a differential operator (or other). There is no reason for task and capability names to coincide, but a straightforward transformation ensures they do, making subsequent manipulations simpler.

   Example 2. Consider the situation where a capability exists, which knows how to integrate univariate equations. The description of the capability in the MSDL is likely to be expressed in terms of anonymous functions as:

   $$R = \int \lambda : x \to f(x)\mathrm{dx} \qquad (1)$$

   here x is an anonymous variable and its name is independant on the meaning. The approach we use is to normalise the variables so that the normalised variable is dependant on the position in the document, for example if it is the 1st variable encountered in a pre-order traversal of the document tree, it might be normalised to n_1. In which case equation 2 would be normalised to:

   $$R = \int \lambda : n_1 \to f(n_1)\mathrm{d}n_1 \qquad (2)$$

5. Commutative operators offer opportunity for confusion like associative since there will be a number of ways to represent an expression using them. Our solution is to define an ordering on the elements of OpenMath objects, then when a capability is registered, the children of any commutative operations will be stored in order. A similar sorting is performed on task. As long as the sorting is structurally based this means that regardless of the ordering (as long as it is well founded), capability and task will be identical down to their leaves. However there is a problem with identifying variables which

[1] By which we mean consistent variable renaming.

are direct descendants of a commutative operation; we deal with this by constructing equivalence classes of documents that are structurally identical modulo variable differences.

We have defined an ordering on two OpenMath objects `o1` and `o2` as follows:

Ordering 1 OpenMath Ordering

---

if o1 and o2 have different element tags **then**
  `OMI` < `OMF` < `OMSTR` < `OMB` < `OMS` < `OMA` < `OMBIND` < `OMV`
else if if o1 and o2 are `OMI` elements **then**
  order on their content (an integer)
else if o1 and o2 are `OMF` elements **then**
  order on the value of the `dec` or `hex` attribute[2]
else if o1 and o2 are `OMSTR` elements **then**
  order lexicographically on their content (a string)
else if o1 and o2 are `OMB` elements **then**
  order on the base64 (defined in reference [RFC]) content of the `OMB` elements
else if o1 and o2 are `OMS` elements **then**
  order lexicographically on value of the cd attribute followed by the value of the name attribute
else if o1 and o2 are `OMA` elements **then**
  recursively order on the children (in document order)
else if o1 and o2 are both `OMBIND` elements **then**
  recursively order on:
    1) first child
    2) number of variables in the `OMBVAR` child
    3) the third child
else
  o1 and o2 are `OMV` elements, these are treated equally.
end if

An example which displays our method for overcoming the commutativity problem is the following:

Example 3. Consider a capability which can integrate piecewise functions, of the following form:

$$f(x) = \begin{cases} x < a & f(x) = g(x) \\ x \geq a & f(x) = h(x) \end{cases} \tag{3}$$

where the inputs are a, f, g, h. The restriction given by equation 3, where the right hand side is a `piecewise` construct, may be given as a pre-condition specified by the capability. Perhaps a task has specified that f is of the form:

$$f(x) = \begin{cases} x \geq a & f(x) = h(x) \\ x < a & f(x) = g(x) \end{cases} \tag{4}$$

[2] we need not be concerned with the incomparability of floating point values as these are IEEE floating point values and thus a finite subset

This specifies the same condition, but in a different form i.e. `piecewise` is commutative. The precondition must be normalised, this results in the children of the `piecewise` descendant of the `pre-condition` element being ordered using the ordering 1. We see that (in this ordering),

```
<OMS cd="relation1" name="lt"/>
```

is smaller than

```
<OMS cd="relation1" name="geq"/>
```

and thus the *part* of the function $f(x)$:

$$x < a \quad f(x) = g(x)$$

is less than

$$x \geq a \quad f(x) = h(x)$$

So the normal form (before variable normalisation) for $f(x)$ is that given by equation 3.

4 Dealing with inputs and outputs

Users must necessarily specify inputs and outputs to any problem they wish to solve. This is so that the variables can be given names and referenced in the pre and post conditions. The user will know what type of inputs they have, so we can require them to provide the types. However the user will not know the name that the service uses or the order that the service specifies them in. This gives us a combinatorial (in the number of inputs/outputs with the same signatures) number of possible orderings for the inputs and outputs. Another way to deal with this problem is to impose a restriction on the user; that is to fix the names of the variables, or the order in which they occur. These two approaches need not be incompatible, as we may keep trying different orderings until a match has been found, or the required number of matches have been found. This may be made part of the normalisation scheme by adding a conjunction of set inclusions to the pre-conditions for the inputs, and post-conditions for the outputs. With each conjunction of set inclusions, a different order of *normalised* names must be forwarded to the α renaming process.

Example 4. If the MSDL of the query specifies the following:

inputs:	$x \in \mathbb{Z}$	**outputs:**	$R \in \mathbb{Z}$
	$y \in \mathbb{Z}$		
	$z \in \mathbb{Q}$		
pre-condition:	$P_{pre}(x, y, z)$	**post-condition:**	$P_{post}(x, y, z, R)$

where x, y, z are the names of the inputs, R is the name of the output and P_{pre}, P_{post} are predicates specifying the pre and post conditions respectively. This will be normalised to the following pair of conjunctions:

pre-condition: $n_1 \in \mathbb{Z} \wedge n_2 \in \mathbb{Z} \wedge n_3 \in \mathbb{Q} \wedge P_{pre}(n_1, n_2, n_3)$

and

post-condition: $n_1 \in \mathbb{Z} \wedge n_2 \in \mathbb{Z} \wedge n_3 \in \mathbb{Q} \wedge \tilde{n}_1 \in \mathbb{Z} \wedge P_{post}(n_1, n_2, n_3, \tilde{n}_1)$

where n_1, n_2, n_3 and $\tilde{n}_1$ are the normalised forms of x, y, z and R respectively. A second alternative may be appropriate for this query, viz.:

pre-condition: $n_1 \in \mathbb{Z} \wedge n_2 \in \mathbb{Z} \wedge n_3 \in \mathbb{Q} \wedge P_{pre}(n_2, n_1, n_3)$

and

post-condition: $n_1 \in \mathbb{Z} \wedge n_2 \in \mathbb{Z} \wedge n_3 \in \mathbb{Q} \wedge \tilde{n}_1 \in \mathbb{Z} \wedge P_{post}(n_2, n_1, n_3, \tilde{n}_1)$

5 Calculating a Similarity Measure

Once the task description has been normalised, it can be compared with the capability descriptions registered with the broker, with the objective of calculating a similarity value. We denote the pre- and post-conditions of task and capability descriptions by $T_{pre}, T_{post}, C_{pre}, C_{post}$ and express the matching requirement between them as:

$$T_{pre} \Rightarrow C_{pre} \wedge C_{post} \Rightarrow T_{post}$$

That is to say, the pre-conditions of the capability must be satisfied by the pre-conditions of the task and the post-conditions of the task must be satisfied by the post-conditions of the capability. In all the following , we consider pre and post conditions in DNF, so $x \in C_{pre}$ means *x is a conjunct in the DNF for the capability pre-condition.* Superfluous task pre-conditions (capability post-conditions) do not effect whether the function may be performed. However it is necessary that their are no extra capability pre-conditions (task post-conditions) as this might allow the client to provide conditions incompatible with the capability pre-condition (capability post-condition). This may be formalised in the following:

$$\exists x_1 \in C_{pre} \text{ s.t. } \forall y_1 \in T_{pre} \mid y_1 \Rightarrow x_1 \tag{5}$$

and

$$\exists x_2 \in T_{post} \text{ s.t. } \forall y_2 \in C_{post} \mid y_2 \Rightarrow x_2 \tag{6}$$

One way of proceeding is to treat the pre and post conditions separately in order to get two similarity values $\mathcal{S}_{pre}$ and $\mathcal{S}_{post}$. If it so happens that the pre and post conditions are equally important, then the average of these values will provide a good measure for the similarity value, however this will not always be the case, other feasible measures are to weight $\mathcal{S}_{pre}$ and $\mathcal{S}_{post}$ linearly with the number of matching disjuncts in the pre-condition match as opposed to the

post-condition match, this is justified as this will give a number of different ways for the conditions to match.

We shall denote the DNF for C_{pre} (or T_{post}) by $R_1 \vee \cdots \vee R_n$ and for C_{post} (or T_{pre}) by $S_1 \vee \cdots \vee S_{\tilde{n}}$. To calculate a value $\in [0.0, 1.0]$ indicating how well equations 5, 6 are satisfied, we use the formula:

$$similarity = \max_{i=1\cdots n}\{\min_{j=1\cdots\tilde{n}}\{M(R_i, S_j)\}\} \tag{7}$$

where $M(R_i, S_j)$ calculates how well conjuncts R_i and S_j match. We may calculate a value for $M(R_i, S_j)$ as:

$$M(R_i, S_j) = \sum_{k=1\cdots\delta} m(S_j, R_{i,k})\frac{1}{\delta} \tag{8}$$

where δ is the number of terms in R_i, $R_{i,k}$ are terms in R_i and:

$$m(S_j, R_{i,k}) \text{ returns } \begin{array}{l} 1.0 \text{ if } R_{i,k} \text{ matches a term in } S_j, \\ 0.0 \text{ otherwise.} \end{array} \tag{9}$$

The term matches which must be performed in order to evaluate the function given by expression 9 may be achieved in a variety of ways, some more effective than others. Two of these methods *algebraic match* and *value substitution match* are expounded in [LRN]. A third rather simplistic method is simply to perform an equivalence test on the XML. An important principle which must never be disregarded is that the term matches must be relatively cheap to perform, it is clearly ridiculous to perform an equivalence test once for every service (in the repository) which is as expensive as the service to be discovered!

6 Overall Matching Algorithm

Our matchmaking architecture is based around two main algorithms: The first is for registering capabilities in the database, this is detailed in algorithm 2.

Algorithm 2 Register a capability

input: $\mathrm{C_{MSDL}}$ {MSDL of capability}
: $\mathrm{C_{URL}}$ {URL of capability}

$\mathrm{N_{MSDL}} \leftarrow$ Normalise MSDL
{normalise the service MSDL, following section 3}
Store $< \mathrm{N_{MSDL}}, \mathrm{C_{MSDL}}, \mathrm{C_{URL}} >$ tuple in registry database
{we store the normalised form as this will save calculating it every time we have a look up, we must still store the original MSDL (for reference) and the capability URL (for access)}

The algorithm 3 takes as input the MSDL of a task, then traverses the capability descriptions stored in the database, calculating a similarity value for each one.

Algorithm 3 Look up

input: $\mathrm{T_{MSDL}}$ {MSDL of task}
output: Collection of triples, {consisting of:
MSDL and URL of capability,
the similarity value of $\mathrm{T_{MSDL}}$ to the capabilities MSDL}

$\mathrm{T_{N(MSDL)}}$ ←normalised MSDL of the task
ret ←new Collection()
{This will accumulate the values to be returned}
for each entry in the registry database **do**
C_{MSDL} ←this capabilities MSDL
$C_{N(MSDL)}$ ←this capabilities normalised MSDL
C_{URL} ←this capabilities URL
S_{Val} ←similarity($T_{N(MSDL)}, C_{N(MSDL)}$) {The similarity value of the normalised task MSDL and the normalised capability MSDL}
add($< C_{MSDL}, C_{URL}, S_{Val} >$ to ret
{add the capabilities MSDL, URL and the similarity value to the task to the collection to be returned}
end for
sort ret by the similarity values
return ret

7 Future Directions

The above method for semantic matching of mathematical capabilities may be extended in a natural way to discover a composition of capabilities which may be used to perform some task. The algorithm required for calculating the similarity value for conjuncts given by expression (8) requires determining a match between individual terms, this allows us to determine which conditions have and which conditions have not been satisfied (by the capability) or required (by the task). A greedy approach may be used to discover some composition of services where overall the task post-condition is met and whose pre-conditions may be satisfied by a conjunct of the task pre-conditions and post-conditions of other services (whose pre-conditions have been met).

8 Conclusion

In this paper, we look at the issues involved with mathematical service matching. We point out the ambiguities occuring in mathematics and suggest a way in which these may be circumvented by converting expressions occuring in the

describing pre and post conditions into a normal form. We suggest a similarity value which measures how similar two services are to each other and analyse how the pre and post conditions of the task and the capability contribute to this similarity value. Finally we look at composition of services and how this ties into the above work.

References

[DNF] http://www-2.cs.cmu.edu/afs/andrew.cmu.edu/course/15/354/www/Lectures/PLogic.pdf

[LRN] S. Ludwig, O.Rana, W. Naylor, J.Padget, Agent-based Matchmaking of Mathematical Web Services, AAMAS 2005.

[MML] Mathematical Markup Language - MathML http://www.w3.org/TR/MathML2/

[MON] Mathematics on the Net - MONET http://monet.nag.co.uk

[MSL] Mathematical Service Description Language http://monet.nag.co.uk/cocoon/monet/publicdocs/monet-msdl-final.pdf

[MPL] Mathematical Problem Ontology http://monet.nag.co.uk/cocoon/monet/publicdocs/d11.pdf

[MEL] Mathematical Explanation Language http://monet.nag.co.uk/cocoon/monet/publicdocs/monet-explanation.pdf

[OM] The OpenMath Society, The OpenMath Standard, 2002, http://www.openmath.org/standard/om11/omstd11.xml

[DSR] D. Richardson, Some Unsolvable Problems Involving Elementary Functions of a Real Variable, Journal of Computational Logic,1968,33,pgs 514-520

[RFC] http://www.mhonarc.org/~ehood/MIME/2045/rfc2045.html#6.8

[STS] J.H. Davenport: "A Small OpenMath Type System", *SIGSAM Bull.* **34**, 16 (2000) 2.