Efficient Matchmaking and Directory Services

Technical Report No IC/2002/77
Date: 18 Nov 2002

Ion Constantinescu, Boi Faltings

Laboratoire d'Intelligence Artificielle, Department Informatique,
Swiss Federal Institute of Technology, IN (Ecublens), CH-1015 Lausanne, Switzerland.
`{ion.constantinescu,boi.faltings}@epfl.ch`

Abstract. It has been widely recognised that matchmaking is an important component of heterogeneous multiagent systems. Several researchers have developed powerful techniques for the matchmaking problem in general. There are also specific representation of agent capabilities such as DAML-S which provide a more specific framework for matchmaking.
Most approaches to matchmaking have assumed a sequential search for an agent with matching capabilities. This may become intractable when the number of available agents gets large. In this paper, we consider how matchmaking can be developed into agent directories that can be searched and maintained efficiently. Our main contribution is to show how matchmaking with DAML-S specifications can be integrated with efficient methods for searching and maintaining balanced directory trees. We also report on experimental results using an implementation based on generalised search trees.

Keywords: Matchmaking, directories, indexing, middle-agents

1 Introduction

For distributed systems information sharing is an inherent, core problem. Still most of the systems (DNS, LDAP) focus on distribution or management issues and worry less about the expressiveness of the described entries and the power of the query process. When talking about applications using intelligent agents we consider entities loosely coupled with rich runtime interactions (e.g. coordination, cooperation, negotiation). In this case the focus is not only on being able to determine relevant partners of interactions (which requires expressive descriptions of advertisements and of user request) but also on a finer grained control of the query process, leaving room to smarter behaviour - like the capability of using components slightly different from the ones initially requested or composing several partial results for providing a response to the original query.

This report is organised as follows: in Section 2 we present the current context regarding structural description formalisms. In Section 3 we describe a number of models for service directories (UDDI and DAML-S). Also we introduce a simple Service Description model and a possible mapping to and from UDDI and

DAML-S. Then in Section 4 we show how structural descriptions (in particular OWL Lite) can be encoded numerically and how that encoding can be applied to service descriptions. Next in Section 5 we take a more detailed look at the matchmaking process and we isolate five types of possible complete and partial matches. In Section 6 we put the matchmaking process in the context of large directories where efficiency is an issue, which usually leads to the creation of search structures or indexes. We describe relevant work in the area of multidimensional access methods and in particular the Generalised Search Tree structure (GiST). Also we show how to create using GiST a search tree of Service Descriptions encoded as numeric multidimensional data. Section 7 shows how partially matching services can be composed to fullfill a requested service. In Section 8 we present some implementation details and we report on some experimental results. Finally in Section 9 we draw some conclusions and we talk about possible future directions.

2 Structural Description Formalisms (Context)

Description logic is a well established field in the domain of Knowledge Representation. One important contribution to the field was KL-ONE [2] for which the subsumption mechanism was proved later to be undecidable [19]. Good overviews of systems for knowledge representation with analisys of tractability regarding different combinations of features can be found in [13] and [5]. Work has also been done lately in an approach that also takes into consideration the computational resources required for the implementation of a practical system [14]. In terms of language development there is a trend pushing the more academic approaches like OIL [6] to approaches closer to current web technology like DAML+OIL [3] and now OWL [22] and OWL Lite [21].

2.1 OWL Lite Model

Web Ontology Language (OWL) is the last step in the evolution of a number of description logic languages towards standardisation and industrial use. OWL Lite is a restricted version of OWL but is completely contained by it. As shown in Figure 1 an OWL Lite ontology may include the following elements:

- Class - which can have taxonomic relations with other classes (subClassAs, sameClassAs)
- Property - which can have taxonomic relations with other properties (subPropertyOf, samePropertyAs, inverseOf). A property can also specify a domain of zero or more classes and a range of zero or more classes.
- Restriction - which is defined between a given class (subClassOf) and a given property (onProperty). It can specify "allValuesFrom" or "someValuesFrom" restriction on the range of a given Property. It can also specify cardinality restrictions (minCardinality, maxCardinality, cardinality).
- Instance - which must be a subclass of class Thing. OWL Lite allows the specification of equality and inequality relations between instances (sameIndividualAs, differentIndividualFrom).

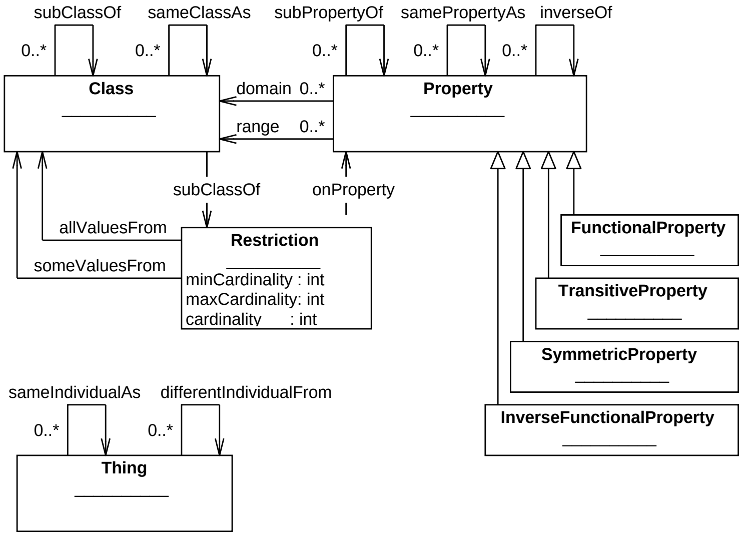


Fig. 1. OWL Lite Model

3 Directories of Services Models (Context)

A good body of work exist in the area of matchmaking including LARKS [20], and the newer efforts geared towards DAML-S [18]. A previous approach [16] for using DAML-S was based on the ConGolog planning framework [10]. Other approaches include the Ariadne mediator [15]. Work has also being done for mapping DAML-S on UDDI [17].

3.1 UDDI Model

The Universal Description, Discovery and Integration (UDDI) [4] is the industry's effort for creating an open specification for directories of service descriptions. It builds on existing technology like XML, SOAP and WSDL.

UDDI v.3 specifies a data model (Figure 2) with 4 levels: business entities which provide services, for which bindings are described in terms of tModels. Note that there is a complete containment for the first three (business, service, binding) but not for the fourth - tModel - which is linked in by reference. This data model can be managed trough an API covering methods for inquiry, publication, security, custody, subscription and value sets.

As it can be seen in Table 1, almost all find_XX methods of the inquiry API (apart find_relatedBusiness which works in a qualitatively different way) can specify keyedRefferences in form of indetifierBags or categoryBags (Figure 2) and can make references to tModels - directly by keys in a tModelBag or indirectly through a find_tModel subquery.

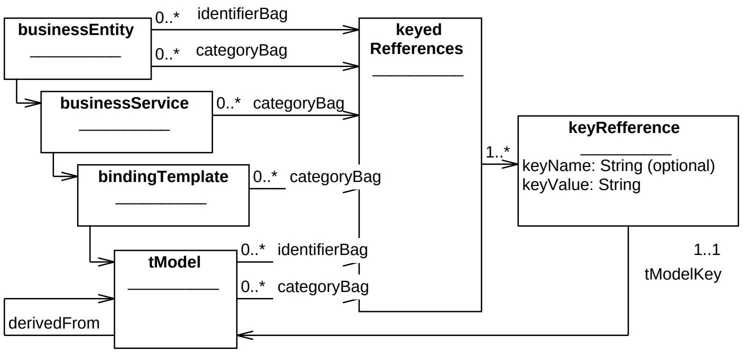


Fig. 2. Model for UDDI v.3

	name	identifierBag	categoryBag	find_tModel	tModelBag	discoveryURLs	find relatedBusiness
find_binding	-	-	x	x	x	-	-
find_business	x	x	x	x	x	x	x
find_service	x	-	x	x	x	-	-
find_tmodel	x	x	x	-	-	-	-

Table 1. UDDI v.3 Inquiry API - find_XX methods

3.2 DAML-S Model

DAML-S is the DAML [3] effort for describing semantic Web Services. DAML-S aims to automate tasks like service discovery, composition, execution and interoperation.

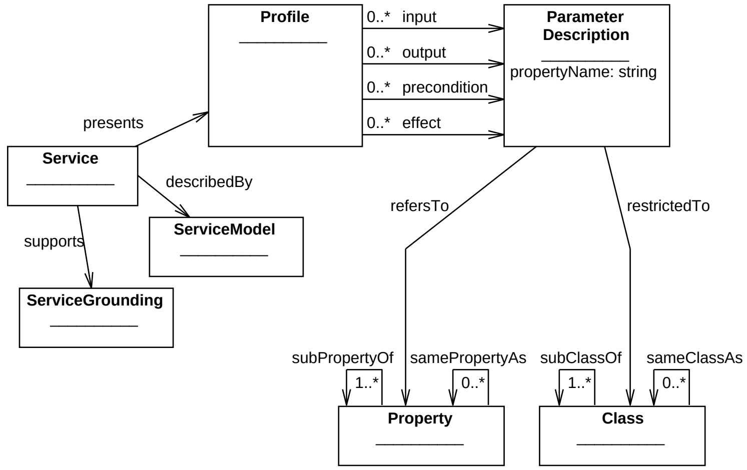


Fig. 3. Model for DAML-S v0.7

In DAML-S v0.7 a Service is defined by three components (Figure 3): a profile, a model and a grounding. The Profile specification was designed to enable automatic matchmaking and discovery by providing information on the capabilities or needs of a service. The Profile class of the DAML-S ontology allows the definition of functional parameters - inputs, outputs, preconditions and effects (IOPEs), and also allows for the definition of non-functional parameters like human contacts, rating, etc.

For any of the IOPEs a DAML-S profile can define zero or more values expressed trough a ParameterDescription construct. Each ParameterDescription is defined by a name, a "refersTo" which specifies the kind of Property that is defined and a "restrictedTo" which specifies the possible values (range) of the defined Property either as a class or as a restriction of a class to a given set of instances. Properties can also be organised as hierarchies trough the "subPropertyOf" attribute.

User services define their profiles by creating ontologies with classes that extend the Profile class or that extend existing well established profiles for a given domain.

3.3 Service Description Model

For the purpose of this paper we assume a simplified model (Figure 4) that covers both UDDI and DAML-S. Our model includes only a top level container

- Service Description which refers to one or more KeyValuePair elements. Each KeyValuePair has exactly one key and a set of one or more values. Keys and values can be expressed in terms of classes and values can be expressed in terms of classes or instances. Classes are organised as hierarchies with equalities using “subClassOf” and “sameClassAs” relations.

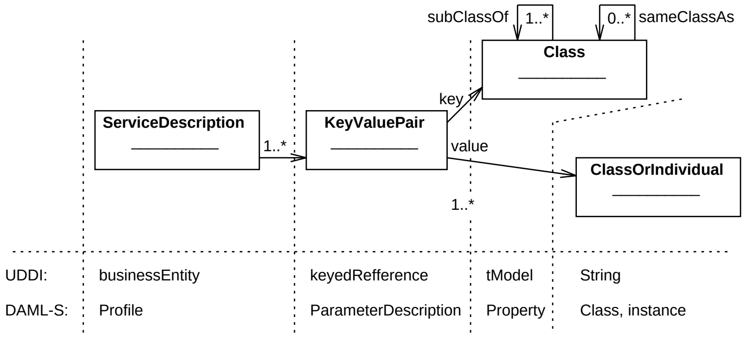


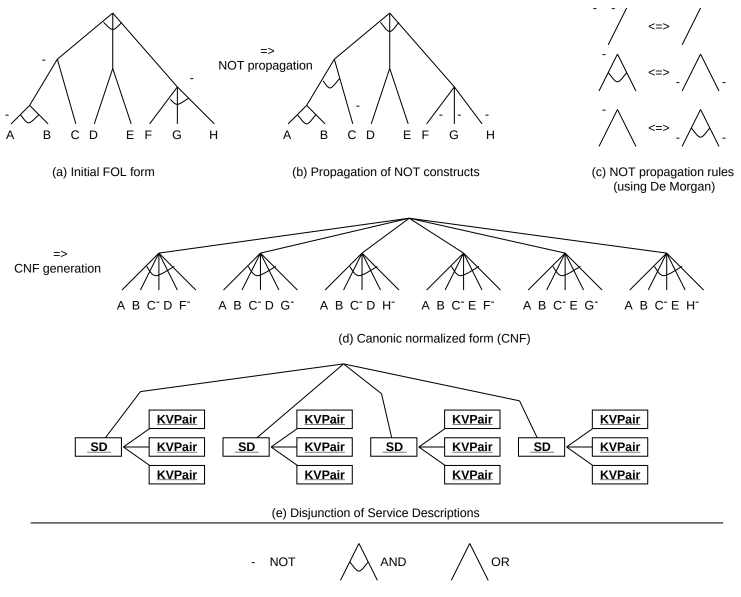
Fig. 4. Proposed Model for Service Description

We show next a possible mapping between the models. An UDDI businessEntity construct or a DAML-S Profile class can be seen as top-level containers similar to our ServiceDescription map. The KeyValuePair is analogous to the UDDI keyedRefference or to the DAML-S ParameterDescription. Since UDDI tModels and DAML-S Properties support inheritance (using derivedFrom and respectively subPropertyOf constructs) we assimilate them with our notion of Classes. Each KeyValuePair also has defined a symbolic name as a string for differentiating between pairs with the same key value.

Finally our model can hold for each key a set of values. They are expressed as either a Class or a Class instance. The equivalent to UDDI would be a string (from a “value set” of possible values per tModel) and the equivalent to DAML-S would be a Class or an instance (a class restricted using a “hasValue” construct). We choose to have a set and not a single value since in the ServiceDescription container keys have to be disjoint. As such “keyValues” of keyedReferences on the same tModel or the “restrictedTo” values of multiple DAML-S PropertyDescription “referring” to the same property are represented in our model by a set of values collated together under the same key - the tModel or respectively the “refersTo” property.

Still as described above in Section 5.2, properties of Service Descriptions describe the “specification” of the service in terms of logical constraints. Currently there is no standard way (e.g. for DAML-S) for specifing such formulas so we propose an aproach based on version of FIPA-SL [8] retstricted to well formed formulas including constructs like **AND**, **OR**, **NOT** and variables.

In our approach Figure 5 we first convert a source Service Description to it’s canonic normalised and we obtain a disjunction of conjunctive service descrip-



(a) Initial FOL form

(b) Propagation of NOT constructs

(c) NOT propagation rules (using De Morgan)

(d) Canonic normalized form (CNF)

(e) Disjunction of Service Descriptions

Fig. 5. Converting Service Descriptions with FOL expressions to CNF

tions as above. Each of the descriptions can contain positive or negative (NOT) terms and variables. Since this generalisation is simple we are not going to make use of it in the rest of this paper but we consider it implicit.

4 Numerically encoding Structural Descriptions

The main barrier towards a fully automated web is the way to describe the meanings of the published information such that machines could interpret it. The Semantic Web community is making efforts in that direction trough a number of languages, specifications and tools (e.g. RDF, RDFS, OWL).

In this paper we are concerned only with a more narrow scope - the representation of class knowledge using a language recently adopted by the W3C - the Web Ontology Language (OWL).

Let's consider the example ontology from the OWL web site [22]. As defined by the specification the root of any OWL class hierarchy is the class Thing. Animal is a subclass of Thing and Female and Male are subclasses of Animal. Animal subsumes also Person. Classes Man and Woman inherit multiply from Male and Person, respectively Female and Person. Class HumanBeing has a "sameClassAs" relation with class Person and is a subclass of Thing.

The main idea of the encoding that we propose is that any class in the hierarchy can be associated with one or more intervals. Intervals can be contained in other intervals but are never overlapping. In Figure 6 we give a possible such encoding for the OWL example ontology.

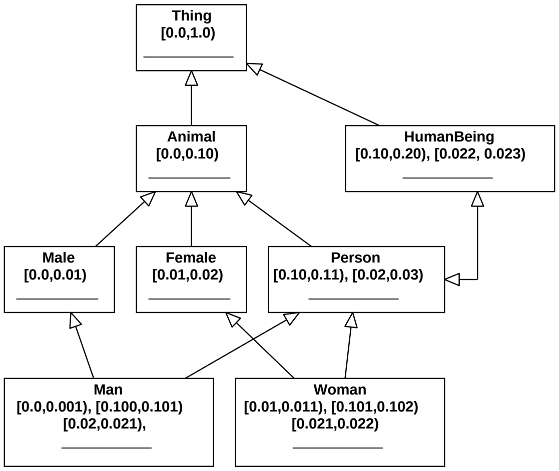


Fig. 6. Example ontology from the OWL web site

Now responding to subsumption questions like Person subsumes Man, Person subsumes Male or HumanBeing subsumes Man is straightforward: one of the intervals of Man [0,0.001), [0.100,0.101), [0.02,0.021) is included in one of the intervals of Person [0.10,0.11), [0.02,0.03); none of the intervals of Male [0,0.01) is included in any of the intervals of Person; finally one of the intervals of Man [0,0.001), [0.100,0.101), [0.02,0.021) is included in one of the intervals of HumanBeing [0.10,0.20),[0.022,0.023).

4.1 Encoding Single Inheritance hierarchies

In this section we address the basic problem of numerically encoding class/property hierarchies where only single inheritance is allowed. We describe only class hierarchies with subClassOf relations because the encoding of property hierarchies is very similar - instead of subClassOf we use subPropertyOf.

Static hierarchies As a first encoding scheme we consider a simple approach where we label in order the leaves of the hierarchy with unit intervals and then we compute intervals of the parent as the union of the intervals of the children. In the example from (Figure7 (a)) E, F are leaves of the hierarchy which we represent as the intervals E [0,1), F [1, 2). B is the parent of both E and F and as such we compute the representation B [0,2) as the union of [0,1) and [1,2). In turn A is the parent of B [0,2) and C [2,5) so we will represent it as A [0,5). We recursively continue to do that until we reach the root of the tree where we stop - in our case we get root [0,7). This encoding completely fullfills our objective stated above: subsumption can now be treated like a simple interval inclusion problem. Let's suppose that we want to determine subsumption relations between query H [3,4) and A [0,5), B[0,2), C[2,5). It is easy to determine that A or C subsume H ([3,4) is included in [0,5) or in [2,5)) but not B ([3,4) is not included in [0,2)). In the wider context this would mean that a services specifying H could be matched with a service specifying A or C but not with with a service specifying B.

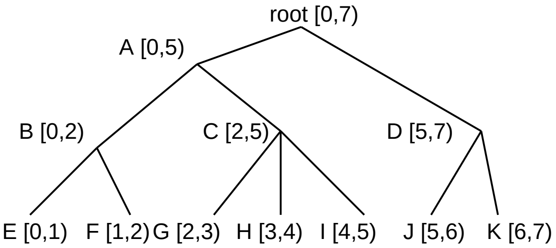


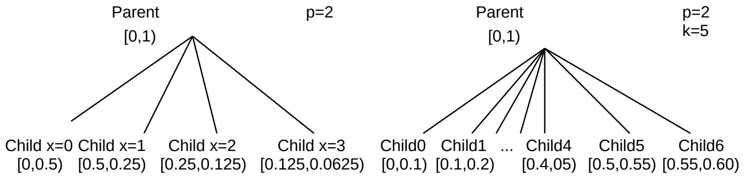
Fig. 7. Encoding of a static hierarchy

Dynamic hierarchies The scheme above is simple, easy to implement and has very loose limitations related to how numeric values are represented by the underlying system. However it applies only to static hierarchies - if additions or deletions are made at different levels at least part of the hierarchy needs to be readjusted. Another drawback is that in the case of deletion it is possible for a new element to have the same representation. Since representations are used to build further structures that may be costly to adapt, we consider next a more complex approach which allows for the representation of dynamic hierarchies.

The dynamic encoding scheme will have two levels: one addressing the child-child relation and one the parent-child relation. First each (direct) parent/child relation encompasses locally an interval between 0 and 1. The children have assigned upon insertion an integer key which is unique and persistent at the parent. Using a dividing function dependent on the key, each child is assigned also an interval. At the second level of the scheme, each parent is assigned also a global interval (e.g. when playing in turn the child role). Local child intervals are then scaled to the global parent interval and global child intervals are in turn computed.

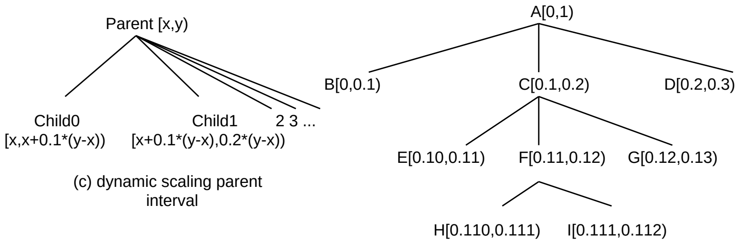
For encoding the local interval we have explored first (Figure 8 (a)) a function involving the division of the length of the previous interval using a given factor p ; this gives us an inverse exponential $invexpP(x) = \frac{1}{p^x}$ (we use p=2). Still as it can be seen also in it's graphic Figure 9 - invexp2 this function rapidly narrows the key interval leading to loss of precision.

Another idea (Figure 8 (b)) for the encoding of the local interval is to linearly divide an interval obtained as before in k parts ; this gives us a linear inverse exponential function:



(a) dynamic invexp2
$1 / p^{x}$

(b) dynamic lin5invexp2
$1 / p^{int(x/k)} + (x\%k)/k * 1 / p^{int(x/k)}$



(c) dynamic scaling parent
interval

(d) example class hierarchy
using scheme in fig (b)

Fig. 8. Encoding of a dynamic hierarchy

$$linKinvexpP(x) = \frac{1}{p^{int(\frac{x}{k})}} + (x\%k) * \frac{1}{k} * \frac{1}{p^{int(\frac{x}{k})}}$$ (we use p=2, k=5).

As it can be seen on Figure 9 the function lin5invexp2 allows for a slower narrowing of the interval and gives more choices for domain specific cases that could use different values for k (e.g. for k=50 graphic lin50invexp2 shows a function which for keys between 0 and 20 is almost linear).

Finally local child intervals are scaled to the global parent intervals and global child intervals are computed (Figure 8 (c)). The scheme uses the start of the global parent interval as an offset to which it is added the size of the parent interval scaled with the local child interval.

Figure 8 (d) shows overall effect of the encoding on a hierarchy where B, C and D are children of A, C has in turn children E, F and G. Determining now subsumption relations between B [0,0.1), C [0.1,0.2) and H [0.110,0.111) is trivial (the interval of H is included in C but not in B so we can easy say that C subsumes H but B doesn't subsume H).

One key question still is the physical limit of such an encoding scheme given by the architecture underlying the implemented system. When using an encoding with a division factor of p and a linear factor of k and considering the smallest positive real number that can be represented on a system to be $L_{minpositive}$ we can compute two limits: the maximum number of entries that we can have on the first level of the hierarchy $MaxClassNo = \log_{p} \frac{1}{kL_{minpositive}}$ and the maximum number of levels that we can have on the first entries of a level $MaxLevelNo = \log_{k} \frac{1}{pL_{minpositive}}$. Note that those are the maximum bounds for

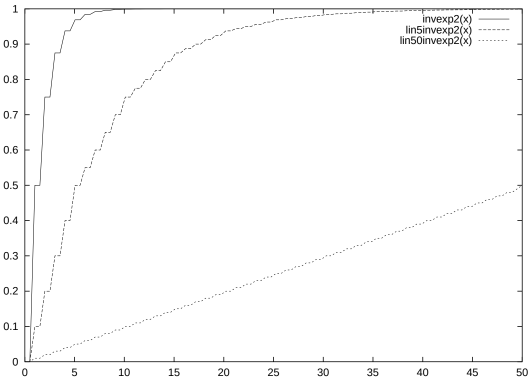


Fig. 9. Evolution of intervals values for division functions for invexpP p=5 and linK-invexpP for p=2, k=5 and for p=2, k=50

classes and levels (since for any level lower that the first we are going to have less classes per level and for any entry different than the first one we are going to have less possible sub levels per entry). For k=5 and p=2 and a system encoding real numbers as 64 bits doubles acordingly to IEEE 754 floating-point "double format" (like for example a J2SE Java^{TM} VM) we have $MaxClassNo = 1071$ and $MaxLevelNo = 462$.

4.2 Encoding Multiple Inheritance relations

In this section we address the problem of encoding class or property hierarchies where multiple inheritance is allowed. Again the encoding is described only in terms of classes since the encoding of properties is very similar (subPropertyOf instead of subClassOf). The main idea of the encoding is to transform multiple inheritance hierarchies into single inheritance hierarchies and then use the technique above (Section 4.1) for representing classes as intervals. For this purpose we augment the initial tree-model with a one-to-many mapping table where for for each multiple inheritance class (node in the tree on the left, Figure 10) we have a mapping to one or more single inheritance classes (tree on the right). We use a nomenclature where single inheritance classes have the same name as the originating multiple inheritance class but followed by the suffix “_i”, *i=0..n* and n is the cardinality of the one-to-many relation (e.g. the multiple inheritance class X maps to the single inheritance classes X_0, X_1, etc.).

We take as an example a multiple inheritance class hierarchy with class A as the root and classes B, C and D as children of A.

First (Figure 10) we consider the case of multiple inheritance of class E from C and D. For each of the superclasses we first add a single inheritance node of E: E_0 subClassOf C_0 and E_1 subClassOf D_0. Then we define the one-to-many relation for E: E=E_0, E_1.

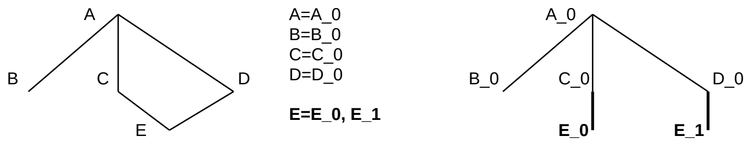


Fig. 10. E subClassOf C and D.

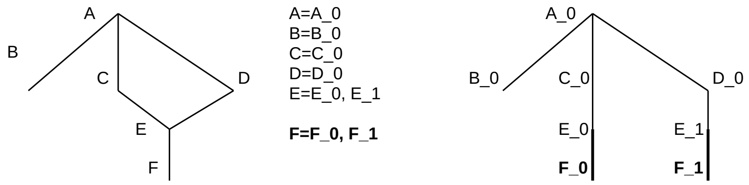


Fig. 11. F subClassOf E.

Next (Figure 11) we examine the case when we add F subClassOf E. E subsuming F, and C and D subsuming E means that C and D must also subsume F. This means that F must create single inheritance nodes for all single inheritance nodes of E. So for each of E_0 and E_1 we add F_0 and F_1 such that F_0 subClassOf E_0 and F_1 subClassOf E_1.

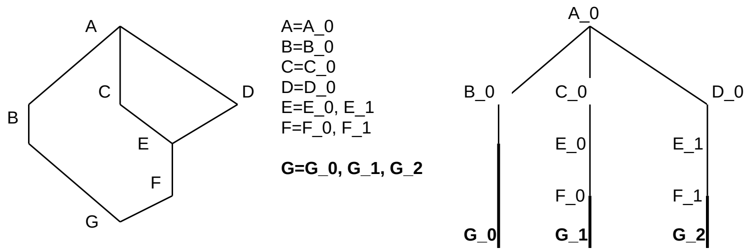


Fig. 12. G subClassOf B, F.

As a third example (Figure 12) we take the case of class G which inherits multiply from B and F. To do that we have to introduce for any mappings of B and F single inheritance nodes G and add those nodes to the mapping of G. This will create G_0 subClassOf B, G_1 subClassOf F_0 and G_2 subClassOf F_1 and the mapping G=G_0, G_1, G_2.

In conclusion for defining subClassOf(X,Y) we have to create the mapping of Y to a number of single inheritance classes, mapping formed by creating for any single inheritance class X_i in the map of X a single inheritance class Y_i in the map of Y:

$$map(Y) = Y\_i \text{ s.t. } \forall X\_i \in map(X), subClassOf(X\_i, Y\_i)$$

But in order to respond to subsumption queries we need also to encode the relations in the single class hierarchy as a intervals. An encoding of the tree using the scheme lin5invexp2 for dynamic hierarchies can be seen in Figure 13. Now responding to subsumption queries like B subsumes G, B subsumes F or C subsumes F is straightforward: G_0 [0,0.01) is included in B_0 [0,0.10), there is no F_i such that F_i is included in B_0 and F_0 [0.10,0.101) is included in C_0 [0.10,0.20).

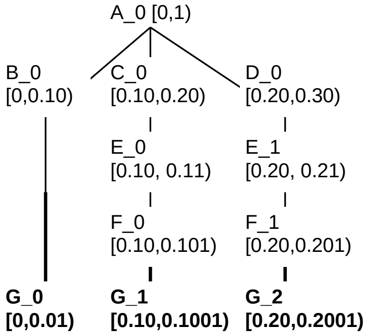


Fig. 13. Numeric encoding of class hierarchy using dynamic lin5invexp2 scheme

So subsumption between classes X and Y boils down to determining if any of the classes in the map of Y is subclass of any of the classes in the map of X:

$$subsumes(X, Y) = \exists X\_i \in map(X) \wedge \exists Y\_i \in map(Y) \wedge$$
$$subClassOf(X\_i, Y\_j)$$

4.3 Encoding Equality Relations

In this section we extend even more our encoding scheme by considering hierarchies which allow for the specification of equality relations between nodes. As described in DAML+OIL axiomatisation [7] a sameClassAs relation is equivalent with two subClassOf relations:

$$sameClassAs(X, Y) \Leftrightarrow subClassOf(X, Y) \wedge subClassOf(Y, X)$$

The same applies also for samePropertyAs and subPropertyOf relations:

$$samePropertyAs(X, Y) \Leftrightarrow subPropertyOf(X, Y) \wedge subPropertyOf(Y, X)$$

We present next only the encoding regarding the sameClassAs relation because the one for samePropertyAs is very similar. Based on the axioms above the encoding of the sameClassAs relations is an extension of the algorithm for multiple inheritance presented previously (Section 4.2). Still we have to avoid the creation of recursive loops that would occur if we would use directly the technique for multiple inheritance. For that the process has to have two steps - first we create two new temporary maps and then we add them to the current maps:

$newmap(X) = X\_i$ s.t. $\forall Y\_i \in map(Y), subClassOf(Y\_i, X\_i)$
$newmap(Y) = Y\_i$ s.t. $\forall X\_i \in map(X), subClassOf(X\_i, Y\_i)$
$map(X) = map(X) \cup newmap(X)$
$map(Y) = map(Y) \cup newmap(Y)$

Let's consider a hierarchy with initially four classes: A as the root and B, C and D as children.

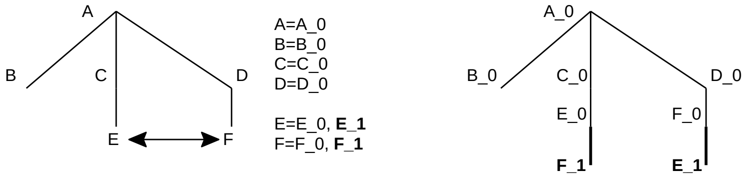


Fig. 14. E sameClassAs F

In a first example (Figure 14) we introduce classes E subClassOf C and F subClassOf D with the map E=E_0 and F=F_0. If we introduce also the relation E sameClassAs F we need to create and add to the map two more single inheritance nodes E_1 and F_1 (E_1 subClassOf F_0 respectively F_1 subClassOf E_0).

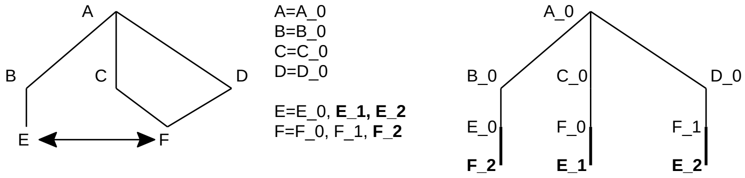


Fig. 15. E sameClassAs F

We consider next (Figure 15) a case where class E is subClassOf B and F has a multiple inheritance from C and D. This will yield in the map E=E_0, F=F_0, F_1. If we introduce the relation E sameClassAs F we have to add to the map of E - E_1 and E_2 and to the map of F - F_2.

Finally we consider a more complex example (Figure 16) where E subClassOf C and D, F subClassOf B and E and G subClassOf D. Introducing the relation F

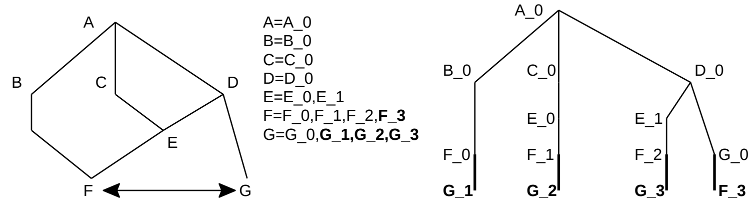


Fig. 16. F sameClassAs G

sameClassAs G yields in four more classes to be added to the single inheritance tree and to the maps of F and G: F_2 and G_1, G_2, G_3.

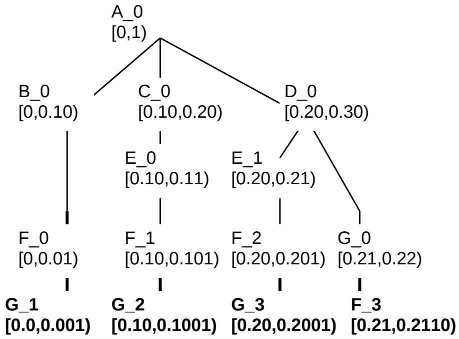


Fig. 17. Numeric encoding of class hierarchy in Figure 16 using dynamic lin5invexp2 scheme

4.4 Numeric Encoding of Structural Descriptions and OWL Lite

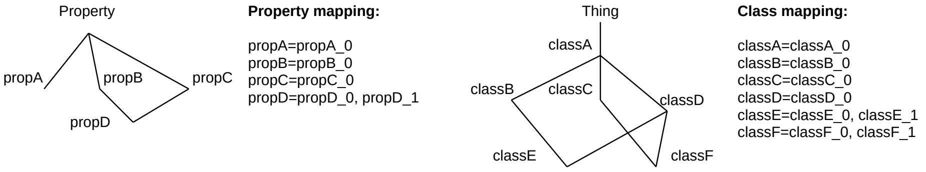
By using the encoding schemes presented previously (Section 4.1, 4.2, 4.3) we are able to cover the majority of the features of the OWL Lite language. Still for a number of features like Restriction "cardinality" and "someValueFrom" or Property "inverseOf", where more approaches are possible, determining the best one definitely requires more work and experimentation on the topic. Also since we consider here only encoding of class-level knowledge we don't address equalities and inequalities for individuals ("sameIndividualAs", "differentIndividualFrom").

4.5 Numeric encoding of service descriptions

Based on the previous Section 3.3 and on Sections 4.1, 4.2, 4.3 the numeric encoding of a service description is now clear: the map between sets of intervals representing properties and sets of intervals representing classes or values can be seens as a set of rectangles in a bidimensional space having on one axis Classes and on the other Properties.

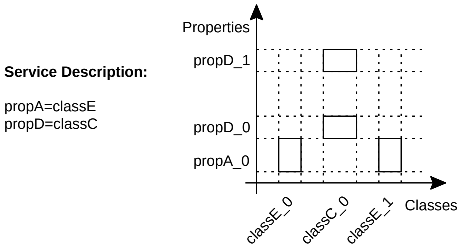
We take as an example the case of a service description with two properties propA and propD which have as values the classes classE and respectively classC (Figure 18 (c)). As it results from Figure 18 (a) propD multiple inherits from propB and propC such that it is going to be represented by a set of two single inheritance classes - propD_0 and propD_0 and their associated intervals. Similarly classE multiple inherits from classB and classD such that it is also going to be represented by two single inheritance classes / intervals - classE_0 and classE_1.

As such the service description above can be represented in a bidimensional space as a set of four rectangles (propA_0 x classE_0, propA_0 x classE_1, propD_0 x classC_0, propD_1 x classC_0).



(a) Property hierarchy

(b) Class hierarchy



(c) Numeric encoding of a service description

Fig. 18. Numeric encoding of a service description

5 Matchmaking

A model accepted [20], [18], [23] by the research community would describe matchmaking as the sum of two processes:

"Matchmaking = Signature Matchmaking + Specification Matchmaking"

5.1 Signature matchmaking

Signature matchmaking deals with determining the class subsumption relation between different class attributes of a query service and a library service. A number of possible signature matches have been identified by Paolucci [18].

5.2 Specification matchmaking

Specification matchmaking deals with determining a constraint subsumption relation between constraints set of a query and a library services. Since general subsumption between constraints (even when reduced to Horn clauses) is intractable, a common approach is to first infer from the constraint set the possible values of the attributes of Q and S (e.g. θ subsumption mechanism used in LARKS [20]). Then the constraint subsumption process boils down to determining a subsumption relation between sets of possible attributes values of Q and S. In a number of possible specification matches have been identified by Zaremski [23].

5.3 An example

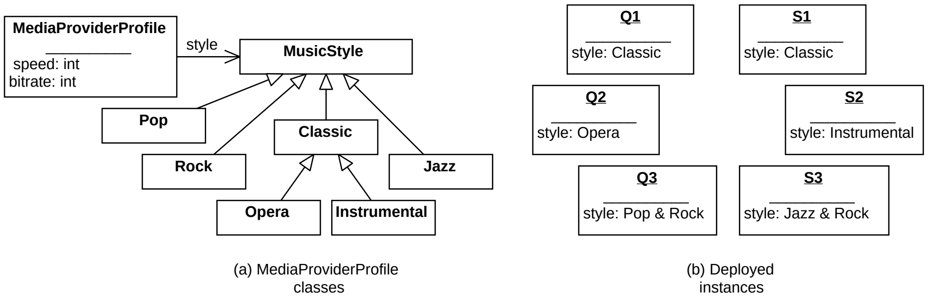


Fig. 19. A network of media providers

Let's consider as an example the problem of matchmaking in a network for media sharing. Media providers advertise their capabilities and consumer user agents define their requests in terms of MediaProviderProfiles (Figure 19 (a)). This class specifies a style of music, a speed and a bitrate.

In the network (Figure 19 (b)) there are a number of such providers and consumers. For example, Q1 is searching for providers of Classic music. S1 will always be able to fullfill such a request since it provides exactly Classical music and the music provided by S2 will also satisfy in all conditions this request since Instrumental music is a particular kind of Classic music. Q2 prefers Opera and his request could be fullfilled by S1. This assumes that either Q2 or S1 would filter out from the Classic music that S1 could provide only the Opera. Q3 prefers

Pop and Rock and it could use S3 but under the constraint that the requests will be only for Rock. This example shows that even when services do not exactly match specification, it may still be possible or necessary to use them in specific instances. Thus, partial matches are also important.

5.4 Supported types of matchmaking

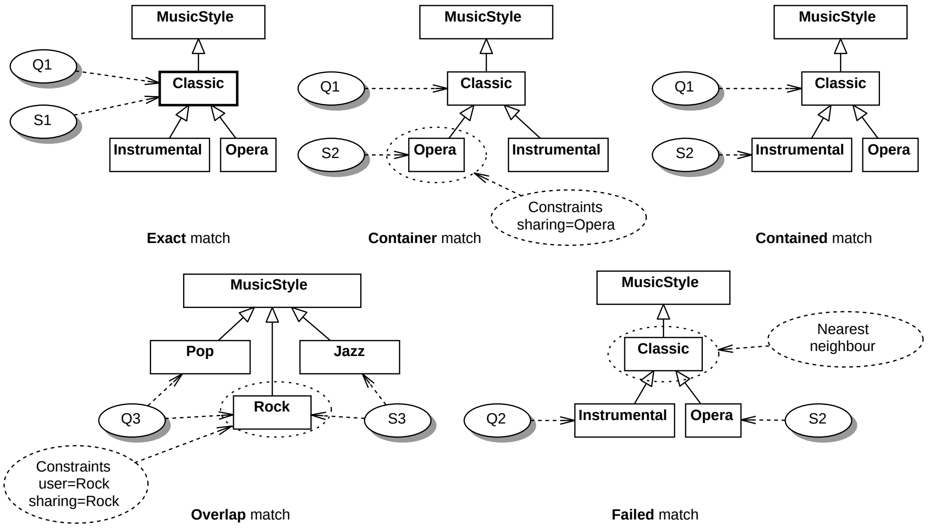


Fig. 20. Signature matching

The outcome of the matchmaking process could be one of the following relations between a query service Q and a library service S (examples in Figure 20 and Figure 21):

- **Exact** - S is an exact match of Q. In our example this is the case of Q1 and S1 for signature matching and Q4 and S4 for specification matching.
- **Contained** - Q is contained in S. In this case S could be always used instead of Q. This is also known as a plug-in match. In our example this is the case for Q1 and S2 for signature matching and Q5 and S5 for specification.
- **Container** - Q contains S. In this case S could be used when additional constraints are specified. In the case of several S's, discrimination between them could be done based on those constraints. For signature this is the case with Q1 and S2. For specification this is the case for Q6 and S6.
- **Overlap** - Q and S have a given intersection. In this case runtime constraints both over Q and S have to be taken into account. For signature this is the case of Q3 and S3. For specification this is the case for Q7 and S7.

- **Failed** - there is no intersection. In this case the system could use a “nearest neighbour” technique to provide services which are as close as possible of Q. For signature this is the case of Q2 and S2. For specification this is the case for Q8 and S8.

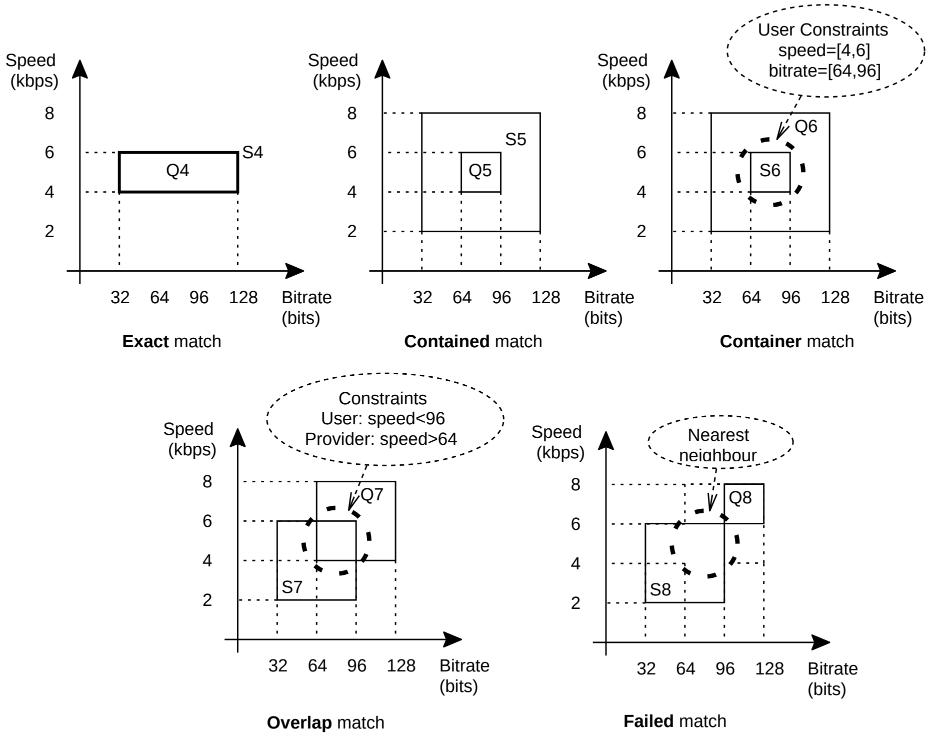


Fig. 21. Specification matching

6 From matchmaking to indexing

Since advertised capabilities or user requests can be expressed as multidimensional data a natural extension would be to assimilate the matchmaking process to a multidimensional access. In a real world environment we could expect that directories with numerous stored advertisements will be created. In this case the matchmaker would have to efficiently deal with data organisation and retrieval. The need for efficient matchmaking would lead to the creation of indexes and search structures for directories.

6.1 Multidimensional Access Methods

There is a lot of work in the database community regarding the indexing and storage of multidimensional objects from rectangles, polygons, CAD drawings to images. A good survey of those methods [9] identifies a number of generic approaches for managing multidimensional data:

- transformation - which deals with managing high dimensional objects by transforming them to simpler representations (e.g. points with more dimensions or sets of intervals in a single dimension). In this category are algorithms like the zkdB$^+$ tree, BANG file or z-ordering.
- overlapping regions - objects are assigned to data buckets which are allowed to mutually overlap. The main problem of this approach is of course that the overlap might increase the number of paths that we have to follow. Example algorithms are the R tree, the R* tree or the skd tree.
- clipping region - objects are assigned to data buckets that are not allowed to mutually overlap. The advantage is obviously in the smaller number of paths that queries have to follow (in the case of point queries a single path from the root to the leaf is sufficient). The main problem with clipping resides with the insertions and deletions phases which have higher overheads due to the fact that objects not fitting exactly in one data bucket have either to be duplicated or to be split between buckets. Examples are the extended KD tree, the R$^+$ tree
- multiple layers - this can be seen as a variant of the overlapping regions approach. The advantage of this technique is the possible higher selectivity during search. Disadvantage are fragmentation, large searches for particular types of queries, ambiguities regarding layer selection and the clustering of data spatially close but placed on different layers.

6.2 Generalised Search Tree (GiST)

As allot of solutions have been proposed for managing multidimensional data work has been done for isolating the common approach that all these solutions take. Hellerstein [12] proposed as an unifying framework the Generalised Search Tree (GiST).

The design principle of GiST starts from the observation that search trees used in databases are balanced trees with a high fanout in which the internal nodes are used as a directory and the leaf nodes point to the actual data. Each internal node holds a key in the form of a predicate P and can hold at the maximum a predetermined number of pointers to other nodes (usually function of system and hardware constraints like filesystem page size). To search for records that satisfy a query predicate Q the paths of the tree that have keys P that satisfy Q are followed. So in GiST terms - any requirement for a general search tree is that *the search key of a given node is a predicate that holds for all the nodes below.*

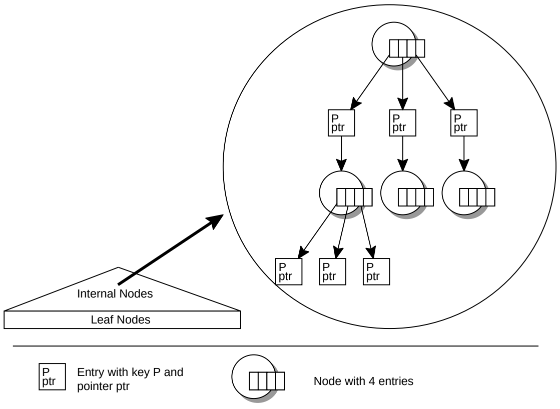


Fig. 22. Generalised Search Tree

A large number of existing tree algorithms can be recasted in terms of GiST: B^+, R tree, R^* tree (by slightly modifying the GiST insertion algorithm), extended KD trees, etc. GiST is quite well adopted by both academic (Postgresql) and industrial (Informix) DB communities for defining access methods to custom data types.

The architecture of GiST is split in two parts: key methods (that have to be implemented for any new application) and tree methods (which are to be provided by toolkits supporting GiST).

Any key has to provide the following methods (there are two more optional methods Compress and Decompress that we don't list here):

- Consistent(E,q) - given an entry E=(p,ptr) and a query predicate q returns false if p q can be guaranteed unsatisfiable, true otherwise.
- Union(P) - given a set P of entries $(p_1,ptr_1)...(p_n,ptr_n)$ returns some predicate r that holds for all entries stored below $ptr_1...ptr_n$.
- Penalty(E_1,E_2) - given two entries $E_1=(p_1,ptr_1)$ and $E_2=(p_2,ptr_2)$ returns a domain specific penalty for inserting E_2 in the subtree rooted at E_1. Typically the penalty metric is some increase of size from $E_1.p_1$ to Union(E_1,E_2).
- PickSplit(P) - given a set P of M+1 entries (p,ptr) splits P into two sets of entries P_1, P_2, each of size at least kM. The choice of the minimum fill factor for a tree is controlled here.

The tree methods trough which an application usually interacts with the GiST tree are:

- Search(q) - returns entries that satisfy the predicate q.
- FindMin(q), Next(q) - efficiently searches a tree when a linear ordering of the domain exists.
- Insert(E) - inserts an entry into the tree.
- Delete(E) - deletes an entry from the tree.

6.3 Utility Service Description Operations

We first introduce a number of utility operations on which we rely later when defining mandatory GiST operations.

- $Equals(Q, S) = (\mid Keys(Q) \mid = \mid Keys(S) \mid) \wedge (\forall K_q \in Q, \exists K_s \in S$ s.t. $K_q \equiv K_s \wedge Q(K_q) \equiv S(K_s))$.

- $Contained(Q, S) = \forall K_q \in Q, \exists K_s \in S$ s.t. $K_q \subseteq K_s \wedge$
  $$\begin{cases} Q(K_q) \subseteq S(K_s), isInput(K_q) \vee isPrecondition(K_q) \\ S(K_s) \subseteq Q(K_q), isOutput(K_q) \vee isEffect(K_q) \end{cases} .$$

  where we assume the existence of four property keys K_{input}, $K_{precondition}$, K_{output}, K_{effect} and we define isInput, isOutput and isEffect as $isInput(K_q) = K_q \subseteq K_{input}$, etc. We define $isPrecondtion(K_q) = \neg isInput(K_q) \vee \neg isOutput(K_q) \vee \neg isEffect(K_q)$ since we assume that all constraints that are non-functional constraints can be treated as preconditions.

- $Container(Q, S) = Contained(S, Q)$.

- $Overlapping(Q, S) = \exists K_q \in Q$ s.t. $\exists K_s \in S, (K_q \subseteq K_s \vee (K_s \subseteq K_q) \wedge (Q(K_q) \subseteq S(K_s) \vee Q(K_s) \subseteq S(K_q))$.

6.4 GiST Related Service Description Operations

In this section we defined the mandatory operations that keys of a GiST tree must implement.

Consistent(Q,S) We define the Consistent(Q,S) predicate to be the same as the Contained(Q',S'). Q' S' are transformation of Q and S by unification and variable substitution. This is equivalent to the θ subsumption used by LARKS [20]:

$$Consistent(Q, S) = Contained(Q', S'), bindings = unify(Q', S'),$$
$$Q' = substitute(bindings, Q), S' = substitute(bindings, S)$$

Union($S_1, ..., S_n$) We use the following formula for defining the union of service descriptions $S_1, ..., S_n$:

$Union(S_1, ..., S_n) = S_u$ s.t. $\forall S_i, i = 1..n, \forall K_i \in S_i,$

$$\begin{cases} S_u(K'_u) = S_i(K_i) \cup S_u(K'_u), \exists K'_u \in S_u, K_i \subseteq K'_u \\ \\ S_u(K_i) = S_i(K_i) \cup S_u(K''_u) \quad \nexists K'_u \wedge \forall K''_u \in S_u, K''_u \subseteq K_i \\ S_u(K''_u) = \emptyset, \\ \\ S_u(K_i) = S_i(K_i), \qquad\qquad otherwise \end{cases}$$

In other words we handle each key K_i of service description S_i in one of the three ways:

- if we find in the current union result S_u a key K'_u in which K_i can be contained we just "add" $S_i(K_i)$ by doing an union with the current $S_u(K'_u)$ values. Note that since keys of a service description are disjoint if K'_u exists it is unique.
- if the above case doesn't occur then for the set of all keys K''_u which are contained in K_i (and which we could track when we search for K'_u) we do two things: create the union of their values including K_i and add it to the result as $S_u(K_i)$; remove from the results all $S_u(K''_u)$.
- if it not the case of any of the above then just add $S_i(K_i)$.

Penalty(Q,S) The definition of this operation is heavily related to the definition of Union($S_1, ..., S_n$) above (6.4) and represents something equivalent with the increase of area between S and Union(Q,S) as suggested by Hellerstein [12]. By area we understand the product between key and value sizes ($\mid K \mid * \mid S(K) \mid$) and by size we understand the sum of interval dimensions ($\mid S(K) \mid = \sum_{\forall I \in S(K)} high(I) - low(I)$). The total area increase is computed as the sum of area increases each key K_q of Q would bring if we would union Q and S.

$$Penalty(Q, S) = \sum d_q \text{ s.t. } \forall K_q \in Q,$$

$$d_q = \begin{cases} \mid K'_s \mid *(\mid Q(K_q) \cup S(K'_s) \mid - \mid S(K'_s) \mid), \\ \qquad\qquad\qquad\qquad \exists K'_s \in S, K_q \subseteq K'_s \\ \\ \mid K_q \mid * \mid Q(K_q) \cup S(K''_s) \mid - \mid K''_s \mid * \mid S(K''_s) \mid, \\ \qquad\qquad\qquad\qquad \nexists K'_s \wedge \forall K''_s \in S, K''_s \subseteq K_q \\ \\ \mid K_q \mid * \mid Q(K_q) \mid \\ \qquad\qquad\qquad\qquad otherwise \end{cases}$$

PickSplit Finally for the PickSplit operation we took an approach inspired by Gutman's R-tree [11] quadratic algorithm. We start by picking two seed entries - the pair of entries with the biggest Penalty between them. We use the seeds as first elements of two groups in which we are going to split the entries of the node. Then we pick in turn from the entries not yet assigned to a group one that has the biggest preference for any of the two groups (maximum difference between the Penalties of adding the entry to each of the two groups).

6.5 Query predicates for GiST

Query predicates for GiST are used when searching for a query Q in the tree. As the query propagates from root to the leaves it evaluates at each node S the contained predicate to determine which search paths (sub-entries) to take. The definitions of these predicates rely on the Service Description utility operations Equals, Contained, Container and Overlapping defined above and on the $level(S)$ function which returns the level in the tree of a given node (0 if S is a leaf of the tree).

- $EqualsP(Q,S) = \begin{cases} Equals(Q,S), & level(S) = 0 \\ Contained(Q,S), & otherwise \end{cases}$
- $ContainedP(Q,S) = Contained(Q,S)$
- $ContainerP(Q,S) = \begin{cases} Container(Q,S), & level(S) = 0 \\ Overlapping(Q,S), & otherwise \end{cases}$
- $OverlappingP(Q,S) = Overlapping(Q,S)$

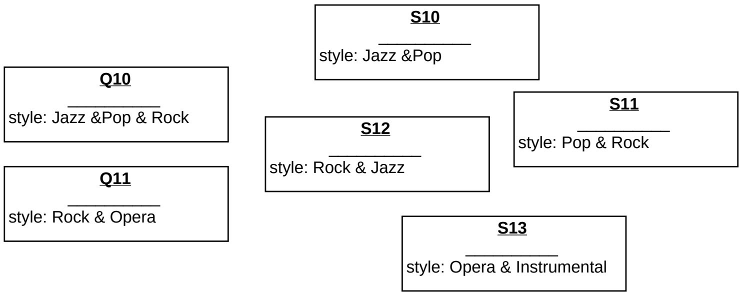
7 Dealing with partial matches

As seen previously (Section 5) when matching queries and services there could be the fortunate case that services match exactly or can be fully substituted (equals or contained match) but there could be also the case that services match only partially (container or overlaps match). In the latter case our goal is to try to fullfill the initial request by composing together the partially matching services.

First we have to provide some more clarifications on the semantics of the considered example: a query represents the styles of music that an user might possibly request and a service advertises the possible styles of music for which it could successfully fullfill a request. As such query $Q10$ will mean that the user could request a *Jazz*, *Pop* or *Rock* melody. $S10$ is able to fullfill requests for *Jazz* or *Pop* styles and $S13$ for *Opera* and *Instrumental* styles.

The process for determining a composed service from a serie of partial matches will be as following:

- when a container or overlap query is submitted the system determines a list of matches. In our example (see Figure 23 (a)) for query $Q10$ and a request for containment matching services - $S10$, $S11$ and $S12$ will be returned; in the case of query $Q11$ and request for overlapping match services $S11$, $S12$ and $S13$ will be returned.



(a) Deployed services

(b) Containment relations

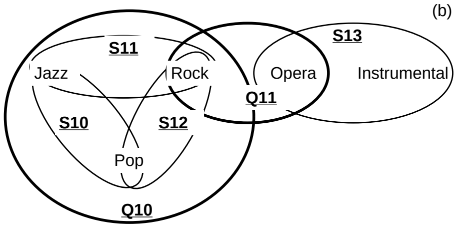


Fig. 23. Composing partial matching (contained or overlapping) services.

- we compute the additional constraints of the query for which certain services can be used. In $Q10$ the user requests $Jazz$ so $S10$ or $S11$ will be used ; if the user requests Pop then $S10$ or $S12$ will be used. Similarly in the case of $Q11$ when the user requests $Opera$ $S13$ will be used and when the user requests for $Rock$ either $S11$ or $S12$ will be used.
- finally for a given query the system returns a software switch that indicates for a given user constraint which services will be possible to use. In the case of $Q10$ we will have:

$$\begin{array}{ll} switch(Q10) & \\ & Jazz \rightarrow S10 \vee S11 \\ & Pop \;\; \rightarrow S10 \vee S12 \\ & Rock \rightarrow S11 \vee S12 \end{array}$$

Similarly in the case of $Q11$ we will have:

$$\begin{array}{ll} switch(Q11) & \\ & Rock \;\;\; \rightarrow S11 \vee S12 \\ & Opera \rightarrow S13 \end{array}$$

8 Implementation and Results

We have implemented a prototype system which has an ontological repository for dealing with class encoding as intervals, an instance repository for dealing with the encoding of values as intervals and a Service Description repository for storing and retreiving numerically encoded services. The Service Description repository is a GiST based tree and can handle basic functionalities like insert, search and delete. We created also a testbed by implementing a number of random generators for ontologies, values and service descriptions.

We have used the testbed to experiment with insertion algorithms and we have made comparable measurements between a tree using for insertion the classic GiST algorithm and another using reinsertion as specified for R^* trees [1].

The results show that the R^* scheme performs slightly better than GiST but also induces bigger fluctuations of resource usage. To give an idea on performance, trees with 10000 entries require around 3 seconds for an insertion and have a depth of 4 levels. Matches are extremely fast in the order of milliseconds.

9 Conclusion and Future work

In this paper we have considered matchmaking of capabilities and requests of distributed systems with focus on UDDI and DAML-S. We have provided a simple model for Service Descriptions and possible mappings from it to the former two. We also proposed a numeric representation that transforms matchmaking into the problem of finding intersections between rectangular structures in hyperspaces.

Then we have used this fact to develop efficient methods for accessing and maintaining directories of services.

Doing that gives us a consistent advance over existing directory systems: first, the query process can take from the beginning into account all possible dimensions of a query and of stored data. In contrast, current systems (e.g. databases, LDAP, etc) usually deal with complex queries over complex data by decomposing them into simpler queries over simpler data and obviously introducing an overhead. The inherent flaw of such an approach is that the real nature (e.g. containment, similarities, etc) of the data cannot be captured.

Second - complex matches (contains, container, overlaps) can be easily computed giving more opportunities to smarter systems to accomplish requested functionality. In particular we show how several partially matching services can be composed for providing the requested service.

Finally, this kind of representation is suitable for creating large indexes - which are going to be mandatory when directories with large amount of stored data are going to be built.

Our contribution is also to provide an implementation (Section 8) where Service Descriptions can be stored and retrieved. We also create a testbed for determining quantitative aspects of the tree system and we report some figures regarding number of entries, tree depth and insertion times. A number of optimisations of the implementation are possible and we consider them as future work.

Future work would also include more experimentation on different methods for organising the tree (e.g. combining machine learning with dimensional estimations) and trying to fine tune the approach for usage in different application domains (small number of classes / large number of instances or vice-versa).

Another promising direction is the distribution and maintenance of the index on several sites. This will have to integrate into the approach also a number of issues specific to distributed systems.

References

1. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM/SIGMOD International Conference on Management of Data*, 1990.
2. R.J. Brachman and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.
3. DAML Consortium. Darpa Agent Markup Language Web Site, http://www.daml.org.
4. UDDI Consortium. Universal Description, Discovery and Integration Web Site, http://www.uddi.org.
5. Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Reasoning in description logics. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 191–236. CSLI Publications, Stanford, California, 1996.
6. D. Fensel, I. Horrocks, F. Van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a nutshell. In Springer-Verlag, editor, *R. Dieng et al., editors, Knowledge Acquisition, Modeling, and Management, Proceedings of the European Knowledge Acquisition Conference (EKAW-2000)*, pages 75–102, Berlin, 2000.
7. Richard Fikes and Deborah L. McGuinness. An axiomatic semantics for RDF, RDF schema, and DAML+OIL. Technical Report KSL Technical Report KSL-01-01, Knowledge Systems Laboratory, Stanford University, 2001.
8. FIPA. FIPA SL Content Language Specification (00008). Technical report, Foundation for Intelligent Physical Agents, 2002.
9. Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
10. Giuseppe De Giacomo, Yves Lesperance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.
11. A. Gutman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD Conf.*, 1984.
12. Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proc. 21st Int. Conf. Very Large Data Bases, VLDB*, pages 562–573. Morgan Kaufmann, 11–15 1995.
13. Bernhard Hollunder, Werner Nutt, and Manfred Schmidt-Schaus. Subsumption algorithms for concept description languages. In *European Conference on Artificial Intelligence*, pages 348–353, 1990.
14. I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–264, 2000.

15. Craig A. Knoblock, Steven Minton, Jose Luis Ambite, Naveen Ashish, Ion Muslea, Andrew Philpot, and Sheila Tejada. The Ariadne Approach to Web-Based Information Integration. *International Journal of Cooperative Information Systems*, 10(1-2):145–169, 2001.
16. S. McIlraith, T.C. Son, and H. Zeng. Mobilizing the semantic web with daml-enabled web services. In *Proc. Second Intĺ Workshop Semantic Web (SemWeb2́001)*, Hongkong, China, May 2001.
17. Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Importing the Semantic Web in UDDI. In *Proceedings of Web Services, E-business and Semantic Web Workshop*, 2002.
18. Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Semantic matching of web services capabilities. In *Proceedings of the 1st International Semantic Web Conference (ISWC)*, 2002.
19. M. Schmidt-Schauss. Subsumption in KL-ONE is undecidable. In *Proc. of KR-89*, pages 421–431, Boston (USA), 1989.
20. K. Sycara, J. Lu, M. Klusch, and S. Widoff. Matchmaking among heterogeneous agents on the internet. In *Proceedings of the 1999 AAAI Spring Symposium on Intelligent Agents in Cyberspace*, Stanford University, USA, March 1999.
21. World Wide Web Consortium (W3C). Feature synopsis for OWL lite and OWL http://www.w3.org/tr/owl-features/.
22. World Wide Web Consortium (W3C). OWL web ontology language 1.0 reference http://www.w3.org/tr/owl-ref/.
23. Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.