Flexible and Efficient Matchmaking and Ranking in Service Directories

Ion Constantinescu, Walter Binder, Boi Faltings
Ecole Polytechnique Fédérale de Lausanne (EPFL)
Email: firstname.lastname@epfl.ch

Abstract

Service directories are a key component of distributed systems where shared information must be managed efficiently. For a directory with a large numbers of entries, the result set of a query may be large, too. In this case it is important to order the results according to heuristics and to retrieve them incrementally. Our contribution is an integrated directory system specially adapted to large-scale service discovery and composition. We introduce DirQL, a flexible query language for the matching and ranking of service descriptions. As results are incrementally retrieved, our system is able to lazily compute the result set based on: 1) the organization of the directory as a special balanced search tree that has an extra "intersection" discriminator, 2) a scheme for transforming the original query into one taking into account the tree structure of the directory, and 3) the organization of partial results in a heap structure sorted according to the transformed query. We also report on experimental results regarding the usage of the directory by a composition engine solving randomly generated problems.[1]

1 Introduction

Service composition is an exciting area which has received a significant amount of interest in the last period. Initial approaches to web service composition [17] used a simple forward chaining technique which can result in the discovery of large numbers of services. There is a good body of work which tries to address the service composition problem by applying planning techniques based either on theorem proving (e.g., Golog [11, 12]) or on hierarchical task planning (e.g., SHOP-2 [18]). All these approaches assume that the relevant service descriptions are initially loaded into the reasoning engine and that no discovery is performed during composition. Recently, Lassila and Dixit [9] have addressed the problem of interleaving discovery and integration in more detail, but they have considered only simple workflows where services have one input and one output.

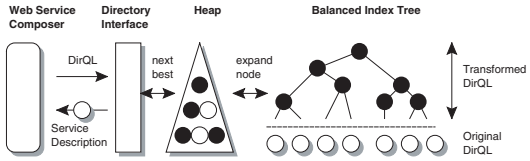

Figure 1. System overview.

The current and future state of affairs regarding web services will be quite different since due to the large number of services and to the loose coupling between service providers and consumers we expect that services will be indexed in directories. Consequently, planning algorithms will have to be adapted to a situation where operators are not known a priori, but have to be retrieved through queries to these directories. Basic planning systems check all the operators in the planning library against the current search state for determining which actions to perform next. In contrast, in the case of service composition, the search state is used to extract the *specification of possible operators*. This specification together with some constraints specific to the composition algorithm is used for formulating queries to the service directory.

In order to support efficient service composition, the directory system has to meet the following requirements:

- **Efficient search:** The internal structure of the directory has to enable an efficient search in the presence of a large number of service descriptions.
- **Flexible matching and ranking:** The query language should allow algorithm-specific heuristics so that the most promising elements of a (possibly large) result set are returned first. However, the internal directory structure should not be exposed to the client.

Our main contribution is a directory system (see Fig. 1) that addresses the two requirements mentioned before in a novel way, first by organizing the directory as a balanced

[1]The work presented in this paper was supported by the European projects KnowledgeWeb (FP6-507482) and DIP (FP6-507483).

Proceedings of the IEEE International Conference on Web Services (ICWS'05)
0-7695-2409-5/05 $20.00 **IEEE**

search tree and secondly by providing DirQL, a flexible language for the matching and ranking of service descriptions. We provide a transformation framework for DirQL expressions that automatically relaxes user queries, enabling the matching and ranking of inner nodes in the directory tree. As internal nodes are expanded, they are stored in a heap structure (sorted according to the ranking), resulting in a best-first directory search.

The current system builds on previous work [3], where performance evaluations pointed up to the necessity of opening the directory to application-specific heuristics tailored to specific service composition algorithms. In turn, supporting ranking of results requires an efficient procedure for evaluating large numbers or search nodes. In our previous system, these issues were addressed by having ranking functions defined through a specialized API and a restricted version of the Java language shipped by the client to the directory for remote evaluation. This approach had a number of disadvantages, the most important being the fact that the user-specified ranking functions could be correctly applied only to leaf nodes of the search tree. Also the system was limited to Java platforms and the query API had only a few basic constructs.

This paper is structured as follows: In the next section we present our approach for flexible selection and ranking of service descriptions. We start by reviewing the current state of the art regarding matchmaking ; then we describe interval constraints, the formalism that we use to model service advertisements and requests. We introduce DirQL, our directory query language. In Section 3 we first show how existing approaches to efficient propositional inference can be applied in our case when multiple constraints have to be matched. This section also gives some insights regarding the organization of the directory as a balanced search tree, where the inner nodes have an "intersection" discriminator in addition to the classic "union" discriminator. Section 4 shows how query transformations combine the flexibility of our query language with the efficiency provided by the internal directory organization. In Section 5 we present an experimental evaluation of our approach on randomly generated composition problems. Finally, Section 6 concludes this paper.

2 Flexible Matchmaking and Ranking

In this section we start by briefly reviewing the current state-of-the-art regarding matchmaking. Then we introduce interval constraints, a supporting formalism which we use for describing service advertisements and requests. Finally, we present DirQL, our language for flexible matching and ranking of service descriptions.

2.1 Matchmaking – Current Approaches

Previous work regarding the matching of software components [19] has considered several possible match types based on the implication relations between preconditions and postconditions of a library component S and a query Q. For example, the **PlugIn** match, one of the most useful match types, is defined as:

$$match_{PlugIn}(Q,S) = \\ (pre_Q \Rightarrow pre_S) \wedge (post_S \Rightarrow post_Q).$$

In LARKS [16] the above condition was adapted, replacing the implication with a more tractable operation, the θ subsumption over sets of constraints ($\preceq_\theta$):

$$match_{PlugIn}(Q,S) = \\ (pre_Q \preceq_\theta pre_S) \wedge (post_S \preceq_\theta post_Q).$$

A set of constraints pre_S θ-subsumes a set of constraints pre_Q ($pre_Q \preceq_\theta pre_S$ or otherwise $pre_Q \sqsubseteq pre_S$ or $pre_Q \Rightarrow pre_S$), if every constraint in pre_Q is subsumed by a constraint in pre_S (similarly for postconditions):

$$pre_Q \preceq_\theta pre_S \Leftrightarrow \\ (\forall C_Q \in pre_Q)(\exists C_S \in pre_S)(C_Q \preceq_\theta C_S).$$

Most recent work regarding matchmaking [13, 10, 3] extended these approaches by using languages based on description logic [1] like OWL (`http://www.w3.org/TR/owl-ref/`) for defining terms of service advertisements and requests.

2.2 Interval Constraints

For describing service advertisements and requests, we use constraints on sets of intervals (possibly generated from class descriptions [3]). A constraint is a special form of first order predicate that universally quantifies over the values of the interval sets. If an interval represents the encoding of a class, the constraint corresponds to a quantification over all the individuals in the class:

$$P(C_1,...,C_n) \Leftrightarrow \\ (\forall x_1 \in C_1) \ldots (\forall x_n \in C_n) P(x_1,...,x_n).$$

We define a number of possible relations between two interval sets C_1 and C_2:

$$C_1 \sqsubseteq C_2 \Leftrightarrow (\forall i_1 \in C_1)\,(\exists i_2 \in C_2)(i_1 \subseteq i_2),$$
$$C_1 \equiv C_2 \Leftrightarrow C_1 \sqsubseteq C_2 \wedge C_2 \sqsubseteq C_1,$$
$$C_1 \dot{\sqcap} C_2 \Leftrightarrow (\exists i_1 \in C_1)\,(\exists i_2 \in C_2)(i_1 \cap i_2 \neq \emptyset).$$

The relation $\neg\dot{\sqcap}$ is the logical negation of $\dot{\sqcap}$ and holds when the argument interval sets are disjoint. We define also two special relations: top $\dot{\top}$ that always holds and bottom $\dot{\bot}$ that never holds. There is a similarity between the θ subsumption relation between sets of clauses and the interval set subsumption relation $\sqsubseteq$. We assume that constraints have unique arities, i.e., constraints with the same name have always the same number of terms.

We define ent, a complex entailment relation between two constraints $P_1(C_{11}, ..., C_{1n})$ and $P_2(C_{21}, ..., C_{2n})$ with the same arity n but possibly different names P_1 and P_2. The predicate $ent(P_1, P_2, op_1, ..., op_n)$ holds if each of the terms C_{1i} and C_{2i} of the two constraints are in the relation specified by the respective operator op_i:

$$ent(P_1, P_2, op_1, ..., op_n) \Leftrightarrow \bigwedge_{i=1}^{n} C_{1i}\ op_i\ C_{2i}$$
$$\text{where } op_i \in \{\equiv, \sqsubseteq, \sqsupseteq, \dot{\sqcap}, \neg\dot{\sqcap}, \dot{\top}, \dot{\bot}\}, 1 \leq i \leq n.$$

We define $notEnt$, a non-entailment relation having semantics in concordance with those of ent: The predicate holds if at least one of the terms C_{1i} and C_{2i} is not in the relation specified by the respective operator op_i:

$$notEnt(P_1, P_2, op_1, ..., op_n) \Leftrightarrow \bigvee_{i=1}^{n} \neg(C_{1i}\ op_i\ C_{2i})$$
$$\text{where } op_i \in \{\equiv, \sqsubseteq, \sqsupseteq, \dot{\sqcap}, \neg\dot{\sqcap}, \dot{\top}, \dot{\bot}\}, 1 \leq i \leq n.$$

Constraints can be grouped in constraint stores. A constraint store $\mathcal{S}$ is logically equivalent to the conjunction of the constraints in the store:

$$\mathcal{S} = \{P_1(C_{11}, ..., C_{1n}), ..., P_k(C_{k1}, ..., C_{km})\} \Leftrightarrow$$
$$P_1(C_{11}, ..., C_{1n}) \wedge ... \wedge P_k(C_{k1}, ..., C_{km}).$$

By combining universal (all) and existential ($some$) quantifiers over a pair of constraint stores $\mathcal{Q}$ and $\mathcal{S}$ we can define eight predicates (e.g., $all_{\mathcal{Q}}all_{\mathcal{S}}$, $all_{\mathcal{Q}}some_{\mathcal{S}}$,..., $all_{\mathcal{S}}all_{\mathcal{Q}}, all_{\mathcal{S}}some_{\mathcal{Q}}$,..., etc). Each of the predicates holds if the two stores contain constraints according to the quantifications $q_{\mathcal{Q}}$ and $q_{\mathcal{S}}$ that are in a relation as defined above by ent:

$$q_1q_2(P_{\mathcal{Q}}, P_{\mathcal{S}}, op_1, ..., op_n) \Leftrightarrow$$
$$((\forall \mid \exists)(P_{\mathcal{Q}} \mid P_{\mathcal{S}}))((\forall \mid \exists)(P_{\mathcal{S}} \mid P_{\mathcal{Q}}))$$
$$(P_{\mathcal{Q}} \in \mathcal{Q})(P_{\mathcal{S}} \in \mathcal{S})\ \ ent(P_{\mathcal{Q}}, P_{\mathcal{S}}, op_1, ..., op_n),$$

where $q_1, q_2 \in \{all_{\mathcal{Q}}, all_{\mathcal{S}}, some_{\mathcal{Q}}, some_{\mathcal{S}}\}$, $store(q_1) \neq store(q_2)$ and where $store(quant_{\mathcal{X}}) = \mathcal{X}$ for $quant \in \{all, some\}, \mathcal{X} \in \{\mathcal{Q}, \mathcal{S}\}$.

We also explicitly define the negation of the quantification predicates with semantics that can be straightforwardly deduced by the application of DeMorgan's laws for quantifier transformation. After applying these transformations (assumed to be already done on the right part of the expression below) the formula can be written in terms of the non-entailment predicate $notEnt$:

$$\neg q_1q_2(P_{\mathcal{Q}}, P_{\mathcal{S}}, op_1, ..., op_n) \Leftrightarrow$$
$$((\exists \mid \forall)(P_{\mathcal{S}} \mid P_{\mathcal{Q}}))((\exists \mid \forall)(P_{\mathcal{Q}} \mid P_{\mathcal{S}}))$$
$$(P_{\mathcal{Q}} \in \mathcal{Q})(P_{\mathcal{S}} \in \mathcal{S})\ \ notEnt(P_{\mathcal{Q}}, P_{\mathcal{S}}, op_1, ..., op_n),$$

where q_1 and q_2 are defined as above and the negation is propagated over the quantifiers using the extended DeMorgan laws: $\neg all \rightarrow some\neg$, $\neg some \rightarrow all\neg$, $\neg ent \rightarrow notEnt$.

We define $count$, a function which returns the cardinality of a set of constraints selected from the constraint store $\mathcal{S}$ according to their entailment relation with constraints in the store $\mathcal{Q}$:

$$count_{\mathcal{Q},\mathcal{S}}(P_{\mathcal{Q}}, P_{\mathcal{S}}, op_1, ..., op_n) =$$
$$\mid \{P_{\mathcal{S}} \in \mathcal{S} : P_{\mathcal{Q}} \in \mathcal{Q}, ent(P_{\mathcal{Q}}, P_{\mathcal{S}}, op_1, ..., op_n)\} \mid.$$

We introduce also $count_{\mathcal{Q}}$ and $count_{\mathcal{S}}$, two functions which return the cardinality of a set of constraints having a given name P from the stores $\mathcal{Q}$ or $\mathcal{S}$:

$$count_{\mathcal{Q}}(P) = \mid \{P(C_1, ..., C_n) \in \mathcal{Q}\} \mid,$$
$$count_{\mathcal{S}}(P) = \mid \{P(C_1, ..., C_n) \in \mathcal{S}\} \mid.$$

2.3 Describing Services by Interval Constraints

We use constraint stores to define service advertisements or service requests. In this paper we consider the latter to be user queries. Input and output constraints are defined over the two elements that describe a parameter – *roles* for semantics and *types* for compatibility of data representations. Preconditions and effects are defined over concepts describing features of the world. The exact semantics of input resp. output parameters are those described in our previous work [6]. Preconditions and effects have semantics similar to those defined by the OWL-S specification (`http://www.daml.org/services`). Service advertisements or service requests are defined trough four kinds of constraints:

- $IN(R, T)$: Defines an input parameter through its role R and type T.
- $OUT(R, T)$: Defines an output parameter through its role R and type T.
- $PRE(F)$: Defines a precondition through the concept F.
- $EFF(F)$: Defines an effect through the concept F.

Let's consider as an example a service description with two input parameters having roles A resp. B and types a1–

a2 resp. b1, two output parameters having roles C resp. D and types c1 resp. d1–d2, two preconditions p1 resp. p2, and one effect g1. This service description would be represented by the following constraint store: $\mathcal{S}$ = { $IN(A, a1 - a2)$, $IN(B, b1)$, $OUT(C, c1)$, $OUT(D, d1 - d2)$, $PRE(p1)$, $PRE(p2)$, $EFF(g1)$ }.

Our formalism is different from the ones reviewed in Section 2.1, as our approach uses a more fine-grained decomposition of service descriptions as sets of individual preconditions or effects. This implies that for the service to work correctly, each of the individual preconditions has to be satisfied. This is more general than the approach of Zaremski or LARKS, where only one precondition as a whole (possibly containing several clauses) has to be satisfied. In particular the θ subsumption of set of clauses in LARKS is equivalent in our case with the inclusion relation $\sqsubseteq$ of a set of intervals from a precondition or effect constraint. In both cases the relation is true if for any component (clause or interval) in the subsumed component set there is a "larger" component in the subsuming component set.

Below we show how the basic **PlugIn** match type is expressed in our formalism. For a query store $\mathcal{Q}$ and a service store $\mathcal{S}$ this match type can be specified as:

$$
\begin{aligned}
&match_{PlugIn}(\mathcal{Q}, \mathcal{S}) = \\
&\quad all_{\mathcal{S}} some_{\mathcal{Q}}(IN_Q, IN_S, \sqsubseteq_{role}, \sqsubseteq_{type}) \wedge \\
&\quad all_{\mathcal{Q}} some_{\mathcal{S}}(OUT_Q, OUT_S, \sqsupseteq_{role}, \sqsupseteq_{type}) \wedge \\
&\quad all_{\mathcal{S}} some_{\mathcal{Q}}(PRE_Q, PRE_S, \sqsubseteq) \wedge \\
&\quad all_{\mathcal{Q}} some_{\mathcal{S}}(EFF_Q, EFF_S, \sqsupseteq).
\end{aligned}
$$

2.4 DirQL – A Query Language for Directories

As directory queries may retrieve large numbers of matching entries (especially when partial matches are taken into consideration), our directory supports sessions in order to incrementally access the results of a query [4]. By default, the order in which matching service descriptions are returned depends on the actual structure of the directory index. However, depending on the service integration algorithm, ordering the results of a query according to certain heuristics may significantly improve the performance of service composition. In order to avoid the transfer of a large number of service descriptions, the pruning, ranking, and sorting according to application-dependent heuristics should occur directly within the directory. As for each service integration algorithm a different pruning and ranking heuristic may be better suited, our directory allows its clients to define custom selection and ranking functions which are used to select and sort the results of a query.

A query consists of provided inputs and required outputs (both sets contain tuples of parameter roles and associated types), provided preconditions and required effects, as well as a custom matching and ranking function. The matching and ranking function is written in the simple, high-level, functional query language DirQL (Directory Query Language). An (informal) EBNF grammar for DirQL is given in Table 1. The non-terminal *number*, which is not shown in the grammar, represents a numeric constant (integer or decimal number) and the non-terminal *word* represents an alphanumeric word of length bigger than 0 used for constraint names.

```
dirqlExpr  : matchExpr | rankExpr | matchExpr rankExpr ;

matchExpr : 'match' boolExpr ;

rankExpr   : 'rank' 'by' ('asc' | 'desc') numExpr ;

boolExpr   : '(' ('and' | 'or') boolExpr+ ')'
           | '(' 'not' boolExpr ')'
           | '(' quantOP word word relOP+ ')'
           | '(' cmpOP numExpr numExpr ')' ;

quantOP    : 'allQallS' | 'allQsomeS' | 'someQallS'
           | 'someQsomeS' | 'allSallQ' | 'allSsomeQ'
           | 'someSallQ' | 'someSsomeQ' ;

relOP      : 'EQUIV' | 'SUBSET' | 'SUPERSET'
           | 'OVERLAP' | 'DISJOINT' | 'T' | 'F' ;

cmpOP      : '<' | '>' | '<=' | '>=' | '==' | '!=' ;

numExpr    : '(' ('+' | '*') numExpr numExpr+ ')'
           | '(' ('-' | '/') numExpr numExpr ')'
           | '(' ('max' | 'min') numExpr+ ')'
           | '(' 'if' boolExpr numExpr numExpr ')'
           | '(' 'count' word word relOP+ ')'
           | '(' ( 'countQ' | 'countS' ) word ')'
           | number ;
```

Table 1. A grammar for DirQL.

The semantics of the boolean expressions `allQallS`, etc., and their negations `(not (allQallS ))`, etc., and of the numeric functions `count`, `countQ`, and `countS` are those defined in Section 2.2 for the predicates $all_{\mathcal{Q}}all_{\mathcal{S}}$, etc., for their negated versions, and for the functions $count_{\mathcal{Q},\mathcal{S}}$, $count_{\mathcal{Q}}$ and $count_{\mathcal{S}}$. The seven relation specifiers `EQUIV`, `SUBSET`, `SUPERSET`, `OVERLAP`, `DISJOINT`, `T`, and `F` correspond to the seven operators $\equiv$, $\sqsubseteq$, $\sqsupseteq$, $\dot{\sqcap}$, $\neg\dot{\sqcap}$, $\dot{\top}$, $\dot{\bot}$.

3 Efficient Directory Search

Our approach for efficiently matching constraint stores has similarities with the approach for efficient propositional inference initially proposed by [15] and then developed and extended by [7, 14, 2]. They proposed a compilation technique where a generic propositional formula is approximated by two Horn formulas $\underline{\Sigma}$ and $\overline{\Sigma}$ that satisfy the following:

$$\underline{\Sigma} \models \Sigma \models \overline{\Sigma} \text{ or equivalently } \mathcal{M}(\underline{\Sigma}) \subseteq \mathcal{M}(\Sigma) \subseteq \mathcal{M}(\overline{\Sigma}).$$

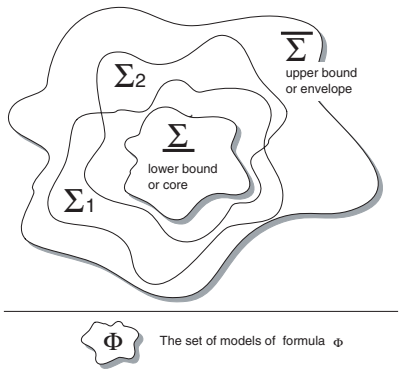

Figure 2. Formula approximations for fast inference.

In the literature $\underline{\Sigma}$ is called the *Horn lower bound* or *core* of Σ while $\overline{\Sigma}$ is called the *Horn upper bound* or *envelope* of Σ. As it can be also seen in Fig. 2, $\underline{\Sigma}$ is a *complete* approximation of Σ while any of the models of the core (lower bound) formula is also a model of the original formula. Conversely the envelope (upper bound) is a *sound* approximation of the original formula as any model of the original formula is also a model of the envelope.

Interval sets in constraints can be also seen as sets of models of a propositional formulas. Interval sets from different constraints can be seen as different formulas (e.g. Σ_1, Σ_2) and can be approximated using the same technique as in the case of Horn clauses (see Fig. 2): their union can be considered an upper bound and their intersection can be considered a lower bound. Before testing individual matching conditions between a query Γ and existing entries Σ_1 resp. Σ_2, the upper and lower bounds can be checked first whether they satisfy a relaxed but necessary version of the original condition, also called pruning condition. If the query description does not match the bound under the relaxed condition, it will certainly not match the original description either. I.e., further tests can be pruned. If the query matches the bound then further tests may be needed, for example on more refined bound approximations.

As an example, let's consider that given an interval Γ we want to determine which of several interval sets Σ_x, $x = 1, 2, ...$ satisfy $\Gamma \sqsubseteq \Sigma_x$. If the intervals Σ_x are bounded by $\underline{\Sigma}$ and $\overline{\Sigma}$, we can use $\Gamma \sqsubseteq \overline{\Sigma}$ as a pruning condition. If this condition does not hold, without further tests we can infer that no interval Σ_x satisfies the original query. If the condition holds, further tests are necessary.

In Fig. 3 we list all other possible inclusion implications between two interval sets Γ and Σ and implicitly the required pruning conditions for $\underline{\Sigma}$ and $\overline{\Sigma}$. We considered five possible set relations: equivalence $\equiv$, subset $\sqsubseteq$, superset $\sqsupseteq$, overlapping sets $\dot{\sqcap}$ and disjoint sets $\neg\dot{\sqcap}$. For the case where no particular relation could be deduced, we have used the

Positive relations ($ent(...)$).

$\Gamma \equiv \Sigma$	$\Rightarrow$	$\Gamma \sqsubseteq \overline{\Sigma}$	$\Gamma \equiv \Sigma$	$\Rightarrow$	$\Gamma \sqsupseteq \underline{\Sigma}$
$\Gamma \sqsubseteq \Sigma$	$\Rightarrow$	$\Gamma \sqsubseteq \overline{\Sigma}$	$\Gamma \sqsubseteq \Sigma$	$\Rightarrow$	$\top$
$\Gamma \sqsupseteq \Sigma$	$\Rightarrow$	$\Gamma \dot{\sqcap} \overline{\Sigma}$	$\Gamma \sqsupseteq \Sigma$	$\Rightarrow$	$\Gamma \sqsupseteq \underline{\Sigma}$
$\Gamma \dot{\sqcap} \Sigma$	$\Rightarrow$	$\Gamma \dot{\sqcap} \overline{\Sigma}$	$\Gamma \dot{\sqcap} \Sigma$	$\Rightarrow$	$\top$
$\Gamma \neg\dot{\sqcap} \Sigma$	$\Rightarrow$	$\top$	$\Gamma \neg\dot{\sqcap} \Sigma$	$\Rightarrow$	$\Gamma \neg\dot{\sqcap} \underline{\Sigma}$

We apply $(A \Rightarrow B) \leftrightarrow (\neg B \Rightarrow \neg A)$ and get:

Negative relations ($notEnt(...)$).

$\neg(\Gamma \equiv \Sigma)$	$\Rightarrow$	$\neg(\bot)$	$\neg(\Gamma \equiv \Sigma)$	$\Rightarrow$	$\neg(\bot)$
$\neg(\Gamma \sqsubseteq \Sigma)$	$\Rightarrow$	$\neg(\bot)$	$\neg(\Gamma \sqsubseteq \Sigma)$	$\Rightarrow$	$\neg(\mathbf{\Gamma} \sqsubseteq \mathbf{\underline{\Sigma}})$, $\neg(\Gamma \equiv \underline{\Sigma})$
$\neg(\Gamma \sqsupseteq \Sigma)$	$\Rightarrow$	$\neg(\mathbf{\Gamma} \sqsupseteq \mathbf{\overline{\Sigma}})$, $\neg(\Gamma \equiv \overline{\Sigma})$	$\neg(\Gamma \sqsupseteq \Sigma)$	$\Rightarrow$	$\neg(\bot)$
$\neg(\Gamma \dot{\sqcap} \Sigma)$	$\Rightarrow$	$\neg(\bot)$	$\neg(\Gamma \dot{\sqcap} \Sigma)$	$\Rightarrow$	$\neg(\mathbf{\Gamma \dot{\sqcap} \underline{\Sigma}})$, $\neg(\Gamma \sqsupseteq \underline{\Sigma})$
$\neg(\Gamma \neg\dot{\sqcap} \Sigma)$	$\Rightarrow$	$\neg(\Gamma \neg\dot{\sqcap} \overline{\Sigma})$	$\neg(\Gamma \neg\dot{\sqcap} \Sigma)$	$\Rightarrow$	$\neg(\bot)$

Figure 3. Selection criteria and required pruning conditions for $\underline{\Sigma}$ and $\overline{\Sigma}$ (right side of the implications).

truth symbol $\top$ (i.e., to make the implication a tautology).

The lower table in Fig. 3 is for the case when the match predicate apears in the query formula negated (e.g., in the body of the $notEnt(...)$ predicate). For determining the pruning conditions for this case he have used the fact that $A \Rightarrow B$ is logically equivalent to $\neg B \Rightarrow \neg A$ and the implications of positive relations between Γ and $\underline{\Sigma}$, $\overline{\Sigma}$ previously determined.

We use the individual pruning conditions above in order to determine pruning forms of the predicates $ent(...)$ and $notEnt(...)$. This is done by replacing in the formula each of the individual tests op_i by a relaxed version accordingly to the right side of the upper left table. Also the second operand is not to be the value from the original constraint but rather its upper approximation $\overline{\Sigma}$. If any of the relaxed tests fail then the predicate will be necesarly false also for the original constraint.

Equivalently for $notEnt(...)$ we generate pruning conditions accordingly to the right side of the lower right table that uses lower bounds $\underline{\Sigma}$. If any of the pruning conditions is true then the tested condition will be necessarly true also for the original constraint and thus the predicate value will be false (due to the negation in $notEnt(...)$).

3.1 Multidimensional Access Methods - GiST

The need for efficient discovery and matchmaking leads to a need for search structures and indexes for directories. We consider service descriptions represented by interval

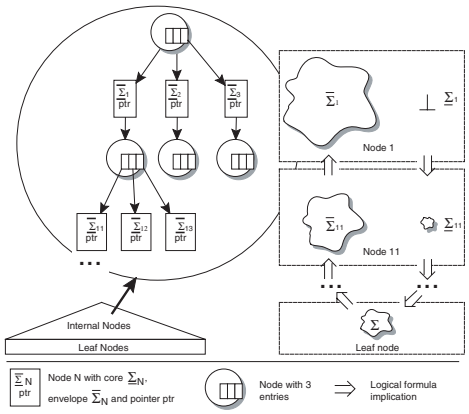

Figure 4. GiST extended with lower bounds.

constraints as multidimensional data and we use techniques related to the indexing of such kind of information for organizing the directory.

The indexing technique that we use is based on the Generalized Search Tree (GiST) structure which was initially proposed as a unifying framework by Hellerstein [8] and later having extensions regarding aspects like concurrency or ranked search. The design principle of GiST arises 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. In the classic GiST each internal node holds a key in the form of a predicate and a number of pointers to other nodes (depending on system and hardware constraints, e.g., filesystem page size). Predicates of inner nodes subsume predicates of all child nodes. To search for records that satisfy a query predicate, only some paths of the tree are followed, those having inner predicates that can satisfy the query being processed. For a given inner node the associated predicate can be seen as an upper bound or envelope (see above $\overline{\Sigma}$) of the predicates in the leaf nodes of the subtree originated at the node.

3.2 Using Lower Bounds in GiST

Our approach extends the basic GiST framework by associating to each inner node a second predicate that is subsumed by all values below. This new predicate can be seen as a lower bound or core (see above $\underline{\Sigma}$) of the predicates in the leaf nodes of the subtree originated at the node defining the predicate. Core predicates $\underline{\Sigma}$ are required for pruning conditions that include negative entailment tests ($notEnt(...)$).

As it can be seen in the example in Fig. 4, while the size of envelope predicates normally increases as they are closer to the root of the tree, normally core predicates become smaller or even empty ($\perp$) as they approach the root.

For inner nodes the constraints in the core are computed as an intersection of the constraints in the cores of children nodes. In turn this is done by intersecting the intervals sets of fields with the same index. Conversely constraints in the envelope are constructed by having for the value of each field in the constraint the union of values of the respective fields of constraints in the envelopes of children nodes.

3.3 Best first tree traversal

By default, the order in which matching service descriptions are returned depends on the actual structure of the directory index (the GiST structure discussed before). However, depending on the service integration algorithm, ordering the results of a query according to certain heuristics may significantly improve the performance of service composition. In order to avoid the transfer of a large number of service descriptions, the pruning, ranking, and sorting according to application-dependent heuristics should occur directly within the directory. As for each service integration algorithm a different pruning and ranking heuristic may be better suited, our directory allows its clients to define custom matching and ranking functions which are used to select and sort the results of a query.

While the query is being processed, the visited nodes are maintained in a heap or priority queue, where the node with the most promising heuristic value comes first (see Fig. 1). Always the first node is expanded: If it is a leaf node, it is returned to the client. Further nodes are expanded only if the client needs more results. This technique is essential to reduce the processing time in the directory until the first result is returned, i.e., it reduces the response time. Furthermore, thanks to the incremental retrieval of results, the client may close the result set when no further results are needed. In this case, the directory does not spend resources to compute the whole result set. Consequently, this approach reduces the workload in the directory and increases its scalability. In order to protect the directory from attacks, queries may be terminated if the size of the internal heap or priority queue or the number of retrieved results exceed a certain threshold defined by the directory service provider.

4 Query Transformation

In this section we give an overview of our transformation scheme that integrates the flexibility and transparency provided by the DirQL language with the efficiency provided by the internal directory structures, i.e., the balanced tree and the heap.

Processing a user query requires traversing the GiST structure of the directory starting from the root node. For validating the final result, the original DirQL expression is

applied to leaf nodes of the directory tree, which correspond to concrete service descriptions.

The client defines only the selection and ranking function for leaf nodes (i.e., to be invoked for concrete service descriptions), while the corresponding functions for inner nodes are automatically generated by the directory. The directory uses a set of simple transformation rules that enable a very efficient generation of the selection and ranking functions for inner nodes (the execution time of the transformation algorithm is linear with the size of the query DirQL expression).

If the client desires ranking in ascending order, the generated ranking function for inner nodes computes a lower bound of the ranking value in any node of the subtree; for ranking in descending order, it calculates an upper bound.

The actual transformation rules start by putting the initial formula in a negated normal form where negation appears only in front of a boolean expression. Positive $ent(...)$ expressions are relaxed using the rules in the upper left section of Fig. 3 applied to the upper bound approximation $\overline{\Sigma}$. In the case of negated expressions of the form $notEnt(..)$ the lower right section of the table is used. In inner nodes the $allS$ quantifier is relaxed to $someS$. Numerical expressions are relaxed by having lower or upper bounds propagated using basic interval arithmetic (e.g., $[X_l, X_u] + [Y_l, Y_u] = [X_l + Y_l, X_u + Y_u]$, $[X_l, X_h] - [Y_l, Y_h] = [X_l - Y_h, X_h - Y_l]$, etc.) Lower bounds are computed as low interval ends and upper bounds are computed as high interval ends. The numeric ranking functions use the core $\underline{\Sigma}$ for lower numeric approximations and the envelope $\overline{\Sigma}$ for upper numeric approximations. A detailed table of all the transformation rules had to be omitted due to space limitations.

5 Evaluation

The service composition planner used for evaluating our approach is based on forward chaining. It iteratively selects an applicable service S (i.e., all inputs required by S have to be available) and applies it to the current world state. The process terminates, if either the requested functionality is provided (i.e., all required outputs are provided) or no solution could be found. Details of such a service composition algorithm are explained in [6].

In the example in Table 2, a matching and ranking function suited for a service composition algorithm using forward chaining with partial type matches is shown. Our forward chaining approach requires that all inputs needed by the service be provided by the query (and the service has to be able to handle some parameter types of the provided inputs, i.e., the types in the query have to overlap with the ones in the service). The results are sorted in ascending order according to the number of still missing outputs after

```
match (and
          (allSsomeQ IN IN SUBSET OVERLAP)
          (allSsomeQ PRE PRE SUBSET))
rank by asc
  (+
    (- (countQ OUT) (count OUT OUT SUPERSET SUPERSET))
    (- (countQ EFF) (count EFF EFF SUPERSET)))
```

```
match (and
       (someSsomeQ IN IN SUBSET OVERLAP )
       (someSsomeQ PRE PRE SUBSET))
rank by asc
 (+
  (- (countQ OUT) (count OUT OUT OVERLAP OVERLAP))
  (- (countQ EFF) (count EFF EFF OVERLAP )))
```

Table 2. Original and transformed query. Changes are indicated in bold. The bars indicate the bound in use.

application of the service. The code for inner nodes is generated according to the transformation scheme described in the previous section.

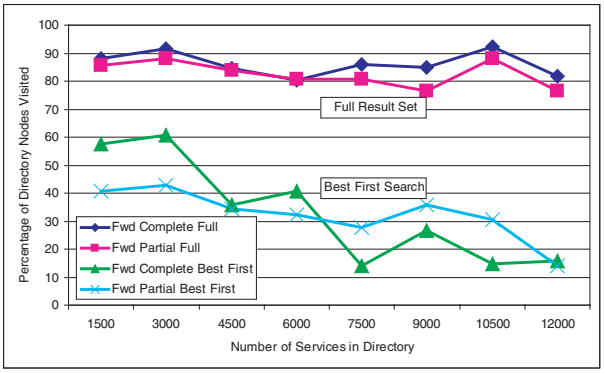

Figure 5. Average percentage of visited directory nodes per query.

We evaluated our approach by carrying out tests on random service descriptions and service composition problems generated as described in [5]. The composition problems were solved using two forward chaining composition algorithms: One that handles only complete type matches and a second one that can compose partially matching services, too (see [6]). Since we were interested in the efficiency of the directory search, we considered as performance metrics the percentage of the total number of index nodes (internal and leaf nodes) evaluated in average for any query submitted by the service composition engine.

We compared two different directory configurations: In the first configuration the directory creates the *full result set* based on the query selection criteria before ranking the re-

sults accordingly to the provided query ranking function. This setting corresponds to our old directory implementation presented in previous work [3]. In the second configuration we evaluated our novel directory that performs a *best-first search* applying the transformed matching and ranking functions to inner nodes, thus lazily creating the result set. For both directories we used exactly the same set of service descriptions and at each iteration we ran the algorithms on exactly the same random problems. The results show (Fig. 5) that the number of directory nodes that are evaluated is consistently smaller in the case of the *best first search* by up to a factor of 6 in the case of the algorithm supporting only complete matches and up to a factor of 3 in the case of the algorithm supporting also partial type matches.

6 Conclusion

In this paper we presented an extensible directory system providing a flexible query language (DirQL) and an efficient way of managing and searching the published service descriptions.

The directory is organized as a special balanced search tree, where nodes contain also an "interesection" discriminator, in contrast to current systems which usually provide only an "union" discriminator. This kind of discriminator is used for early pruning in the case of negated queries and for providing tighter lower bounds in the case of numerical functions. For efficient search, the initial user query is automatically transformed into a query exploiting the internal directory structure (lower and upper bounds). A *best first search* technique is used for the lazy creation of the result set.

Performance measurements with two kinds of composition algorithms based on randomly generated service descriptions and problems confirm that this *best first search* evaluates considerably less directory nodes than our previous directory implementation.

References

[1] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.

[2] M. Cadoli and F. Scarcello. Semantical and computational aspects of horn approximations. *Artif. Intell.*, 119(1-2):1–17, 2000.

[3] I. Constantinescu, W. Binder, and B. Faltings. An Extensible Directory Enabling Efficient Semantic Web Service Integration. In *3rd International Semantic Web Conference (ISWC04)*, Hiroshima, Japan, November 2004.

[4] I. Constantinescu, W. Binder, and B. Faltings. Directory services for incremental service integration. In *First European Semantic Web Symposium (ESWS-2004)*, Heraklion, Greece, May 2004.

[5] I. Constantinescu, B. Faltings, and W. Binder. Large scale testbed for type compatible service composition. In *ICAPS 04 workshop on planning and scheduling for web and grid services*, 2004.

[6] I. Constantinescu, B. Faltings, and W. Binder. Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS-2004)*, San Diego, CA, USA, July 2004.

[7] G. Gogic, C. H. Papadimitriou, and M. Sideri. Incremental recompilation of knowledge. In *National Conference on Artificial Intelligence*, pages 922–927, 1994.

[8] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proc. 21st Int. Conf. Very Large Data Bases, VLDB*, pages 562–573. Morgan Kaufmann, 11–15 1995.

[9] O. Lassila and S. Dixit. Interleaving discovery and composition for simpleworkflows. In *Semantic Web Services, 2004 AAAI Spring Symposium Series*, 2004.

[10] L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th International Conference on the World Wide Web*, 2003.

[11] S. McIlraith, T. Son, and H. Zeng. Mobilizing the semantic web with daml-enabled web services. In *Proc. Second International Workshop on the Semantic Web (SemWeb-2001)*, Hongkong, 2001.

[12] S. A. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams, editors, *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 482–496, San Francisco, CA, Apr. 22–25 2002. Morgan Kaufmann Publishers.

[13] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proceedings of the 1st International Semantic Web Conference (ISWC)*, 2002.

[14] B. Selman and H. Kautz. Knowledge compilation and theory approximation. *J. ACM*, 43(2):193–224, 1996.

[15] B. Selman and H. A. Kautz. Knowledge compilation using horn approximations. In *National Conference on Artificial Intelligence*, pages 904–909, 1991.

[16] 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.

[17] S. Thakkar, C. A. Knoblock, J. L. Ambite, and C. Shahabi. Dynamically composing web services from on-line sources. In *Proceeding of the AAAI-2002 Workshop on Intelligent Service Integration*, pages 1–7, Edmonton, Alberta, Canada, July 2002.

[18] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S web services composition using SHOP2. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, 2003.

[19] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.