Selection and Ranking of Propositional Formulas for Large-Scale Service Directories

Abstract

When composing services using planning techniques, operators have to be incrementally discovered from a directory of service advertisements according to a service request generated from the current composition state. Since service directories can contain large numbers of advertisements and the service request can be complex, it is important that the discovery process in the directory is highly efficient.

Our contribution is a directory system that is able to represent service advertisements and requests as propositional formulas and to provide a flexible query language allowing complex selection and ranking expressions. The internal structure of the directory enables efficient selection and ranking in the presence of a large number of services thanks to its organization as a balanced tree with an extra "intersection" discriminator. In order to optimally exploit the index structure of the directory, a transformation scheme is applied to the original query. Experimental results on randomly generated service composition problems illustrate the benefits of our approach.

Introduction

Service composition is an area which has received a significant amount of interest in the last period. Current approaches are based on planning techniques that rely either on theorem proving (e.g., Golog (McIlraith & Son 2002)) or on hierarchical task planning (e.g., SHOP-2 (Wu *et al.* 2003)). 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 (Lassila & Dixit 2004) 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.

However, due to the large number of services and to the loose coupling between service providers and consumers, services are advertised and indexed in directories. Consequently, planning algorithms must 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 a query to the service directory.

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

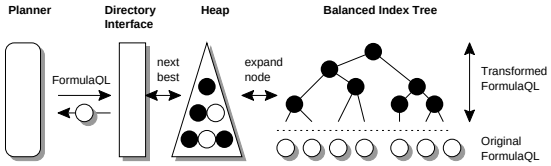

Figure 1: System overview.

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

- **Flexible selection and ranking:** The query language has to support user-defined search 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. Because there is no widely accepted, standard service composition algorithm at the moment, opening the directory for custom heuristics is essential in order to let researchers optimize the directory search for different composition algorithms.
- **Efficient search:** The internal structure of the directory has to enable an efficient search in the presence of a large number of service descriptions.

Our main contribution is a directory system that addresses these two requirements in a novel way, first by organizing the directory as a balanced search tree and secondly by providing FormulaQL, a flexible language for the selection and ranking of propositional formulas (see Figure 1). We provide a transformation framework for FormulaQL expressions that automatically relaxes given query expressions, enabling the 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.

This paper is structured as follows: In the next section we discuss the process of service publication and discovery and introduce our formalism for describing services. In Section "Flexible Selection and Ranking of Propositional Formulas" we present our approach to flexible selection and ranking of propositional formulas. We introduce FormulaQL, our directory query language, and show how existing approaches for efficient propositional inference can be applied in our case.

In Section "Efficient Directory Search" we describe the internal organization of the directory. Section "Query Transformation" explains how query transformations reconcile the flexibility of our query language with the internal directory organization to enable a customized and efficient directory search. In Section "Evaluation" we investigate the performance of our directory for randomly generated service composition problems. Finally, Section "Conclusion" ends this paper.

Large-Scale Service Discovery

The general idea of the discovery process is to select from a potentially large number of Service Advertisement(s) (SA) published in a Service Directory those that fulfill requirements specified by a Service Request (SR). The SR together with a *FormulaQL expression* make up the directory query. A FormulaQL expression may comprise a selection expression, which defines necessary conditions for SA(s) to match the given SR, as well as a ranking expression that specifies the order in which matching SA(s) have to be returned.

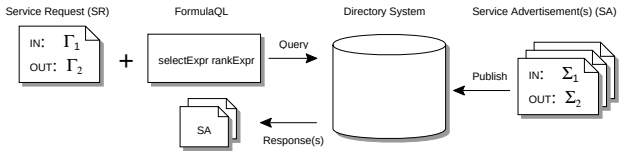

Figure 2: Querying the service directory.

SA(s) and the SR are represented as one or more propositional formulas (for example, $\Gamma_1, \Gamma_2, \Sigma_1, \Sigma_2$ in Figure 2) where each formula is uniquely identified by a keyword (e.g., IN, OUT). As an example, we consider a service that lists flights for Edelweiss, a Swiss company that offers flights to different holiday destinations. The advertisement for this service may contain two formulas: The first formula is identified by the keyword IN and specifies required inputs. The second formula is identified by the keyword OUT and defines possible outputs. As inputs, the service requires flight date, departure airport (e.g., Geneva or Zurich), and arrival airport (e.g., Varna, Heraklion, etc.). As output, the service returns flight number, departure time, and price.

$$\begin{array}{ll} \text{IN} & : flight\_date \wedge (dep\_Geneva \vee dep\_Zurich) \wedge \\ & \quad (arr\_Varna \vee arr\_Heraklion \vee ...) \\ \text{OUT} & : flight\_no \wedge dep\_time \wedge flight\_price \end{array}$$

In terms of expressivity, our formalism is more powerful than the keyword bags used in UDDI[1], the current industry solution for service discovery. Regarding planning languages, it is more powerful than STRIPS (Fikes & Nilsson 1971) and similar to ADL (Pednault 1989). Our formalism does not directly support types, but variable-free first-order logic formulas. Thus, our formalism can express also some restricted Description Logic (Baader & Sattler 2001) statements. Currently Description Logic is used for semantically describing Web Services[2].

[1] http://www.uddi.org/

[2] http://www.swsi.org/

Flexible Selection and Ranking of Propositional Formulas

We define a propositional formula ϕ in the standard way as:

$$\phi = l \mid \neg\phi_1 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2,$$

where l stands for a proposition and formulas can be created from other formulas using the basic logical operators negation $\neg$, conjunction $\wedge$, and disjunction $\vee$.

If we use model-theoretic semantics and define as $\mathcal{M}(\phi)$ the set of satisfying truth assignments of the formula ϕ (i.e., the set of models of the formula), the entailment relation $\Gamma \models \Sigma$ is equivalent to $\mathcal{M}(\Gamma) \subseteq \mathcal{M}(\Sigma)$. I.e., the set of models of Γ is included in the set of models of Σ. If in turn we consider each model as a set of positive propositions and we use the standard set operator $m_1 \subseteq m_2$ for testing whether model m_2 subsumes model m_1, we can specify the entailment relation as:

$$\Gamma \models \Sigma \Leftrightarrow (\forall m_i \in \mathcal{M}(\Gamma))(\exists m_j \in \mathcal{M}(\Sigma))(m_i \subseteq m_j).$$

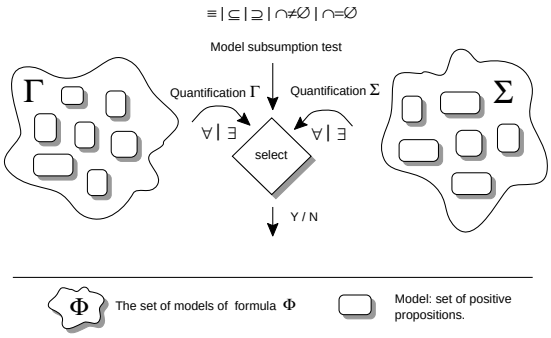

Figure 3: Flexible selection of propositional formulas.

Figure 3 illustrates how we generalize the $\models$ relation, supporting different quantifications over the set of models, as well as several inclusion relations that can be tested between concrete models (sets of positive propositions). For this purpose, we introduce the predicate $select$:

$$select(q_1, q_2, \Gamma, \Sigma, op) \Leftrightarrow$$
$$(q_1\ m_i \in \mathcal{M}(\Gamma))\ (q_2\ m_j \in \mathcal{M}(\Sigma))(m_i\ op\ m_j)$$

where $q_1, q_2 = (\forall \mid \exists)$,
$op = (\equiv \mid \subseteq \mid \supseteq \mid \cap \neq \emptyset \mid \cap = \emptyset)$.

These selection criteria correspond to the Exact, PlugIn, Subsume, and Intersect match types for service advertisements and requests identified by (Paolucci *et al.* 2002; Li & Horrocks 2003; Constantinescu & Faltings 2003).

For determining "how much" a formula Γ entails another formula Σ, we introduce the $rank$ function as follows:

$$rank(\Gamma, \Sigma, op) = \mid \{\ m_i \in \mathcal{M}(\Gamma)\ /\ (\exists m_j \in \mathcal{M}(\Sigma)) \\ (m_i\ op\ m_j)\ \}\ \mid$$

where $op = (\equiv \mid \subseteq \mid \supseteq \mid \cap \neq \emptyset \mid \cap = \emptyset)$.

```
dirqlExpr: selectExpr | rankExpr | selectExpr rankExpr

selectExpr: 'select' boolExpr

rankExpr: 'order' 'by' ('asc' | 'desc') numExpr

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

quantOP : 'allSRallSA' | 'allSRsomeSA' | 'someSRallSA'
        | 'someSRsomeSA' | 'allSAallSR' | 'allSAsomeSR'
        | 'someSAallSR' | 'someSAsomeSR'

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 ')'
        | '(' 'countSR' word ')'
        | '(' 'countSA' word ')'
        | number
```

Table 1: A grammar for FormulaQL.

FormulaQL – A Query Language for Propositional Formulas

When searching a collection of formulas for entailment, the client submits a query consisting of a service request as well as a custom selection and ranking function. The selection and ranking function is written in the simple, high-level, functional query language FormulaQL (Formula Query Language). An (informal) EBNF grammar for FormulaQL 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 a non-empty alphanumeric word used for the names of the keys.

The semantics of the boolean expressions `allSRallSA`, etc., and their negations `(not (allSRallSA ))`, etc., and of the numeric functions `count`, `countSR`, and `countSA` are those defined in the previous section for the predicate $select$ and for the function $rank$. For $select$ and $rank$, the formulas Γ and Σ are retrieved from the service request SR resp. from the service advertisement SA according to the two keys specified as parameters. For example, `(allSRsomeSA IN IN EQUIV)` is equivalent to $select(\forall, \exists, \Gamma, \Sigma, \equiv)$, where $\Gamma = SR(IN)$, $\Sigma = SA(IN)$.

The functions `countSR` and `countSA` return the number of models for a given key in the service request SR resp. in the service advertisement SA. The relation specifiers `EQUIV`, `SUBSET`, `SUPERSET`, `OVERLAP`, `DISJOINT` correspond to the operators $\equiv$, $\subseteq$, $\supseteq$, $\cap \neq \emptyset$, $\cap = \emptyset$. In the case of the operators `T` (true) and `F` (false), the selection function is considered to return always true resp. false. For the operator `T`, the `count` function returns the size of the first argument formula, whereas for the operator `F`, it returns 0.

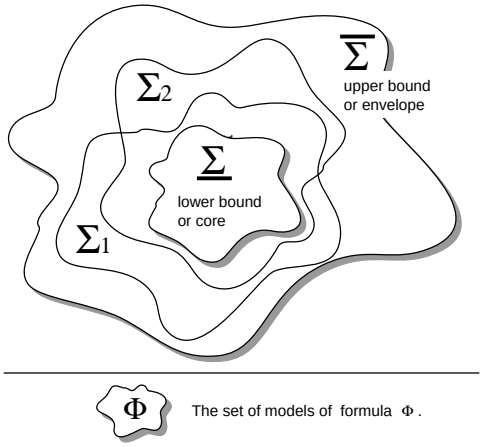

Figure 4: Formula approximations for fast inference.

Efficient Propositional Inference

Our approach for efficiently computing formula entailment is related to the one initially proposed by (Selman & Kautz 1991) and then developed in several other works (Cadoli & Scarcello 2000). They proposed a compilation technique where a generic formula is approximated by two Horn formulas $\underline{\Sigma}$ and $\overline{\Sigma}$ that satisfy the following:

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

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 seen in Figure 4, $\underline{\Sigma}$ 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.

Several formulas (e.g., Σ_1 and Σ_2 in Figure 4) may be approximated by common bounds: The union of their models can be considered an upper bound and the intersection of their models can be considered a lower bound. Hence, before testing individual entailment between Γ and Σ_1 resp. Σ_2, the bounds can be tested as pruning conditions.

As an example, assume that from several service advertisements Σ_x ($x = 1, 2, ...$) bounded by $\underline{\Sigma}$ and $\overline{\Sigma}$, we have to select those that satisfy the entailment $\Gamma \models \Sigma_x$, where Γ is a service request. As a necessary condition for Σ_x to satisfy the entailment, $\overline{\Sigma}$ must satisfy the entailment, too. If this is the case, the individual formulas Σ_x have to be tested for entailment. Otherwise, no further tests are necessary. I.e., the negation of the entailment, $\Gamma \not\models \overline{\Sigma}$, can be used as a pruning condition.

In Figure 5 we list all other possible inclusion implications between a request Γ and an advertisement Σ, as well as the corresponding pruning conditions for $\underline{\Sigma}$ and $\overline{\Sigma}$. We considered five possible set relations: Equivalence $\equiv$, subset $\subseteq$, superset $\supseteq$, overlapping sets $\cap \neq \emptyset$, and disjoint sets $\cap = \emptyset$.

Positive relations ($select(...)$).

$\Gamma \equiv \Sigma$	$\Rightarrow$	$\Gamma \subseteq \overline{\Sigma}$	$\Gamma \equiv \Sigma$	$\Rightarrow$	$\Gamma \supseteq \underline{\Sigma}$
$\Gamma \subseteq \Sigma$	$\Rightarrow$	$\Gamma \subseteq \overline{\Sigma}$	$\Gamma \subseteq \Sigma$	$\Rightarrow$	$\top$
$\Gamma \supseteq \Sigma$	$\Rightarrow$	$\Gamma \cap \overline{\Sigma} \neq \emptyset$	$\Gamma \supseteq \Sigma$	$\Rightarrow$	$\Gamma \supseteq \underline{\Sigma}$
$\Gamma \cap \Sigma \neq \emptyset$	$\Rightarrow$	$\Gamma \cap \overline{\Sigma} \neq \emptyset$	$\Gamma \cap \Sigma \neq \emptyset$	$\Rightarrow$	$\top$
$\Gamma \cap \Sigma = \emptyset$	$\Rightarrow$	$\top$	$\Gamma \cap \Sigma = \emptyset$	$\Rightarrow$	$\Gamma \cap \underline{\Sigma} = \emptyset$

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

Negative relations ($\neg select(...)$).

$\neg(\Gamma \equiv \Sigma)$	$\Rightarrow$	$\neg(\bot)$	$\neg(\Gamma \equiv \Sigma)$	$\Rightarrow$	$\neg(\bot)$
$\neg(\Gamma \subseteq \Sigma)$	$\Rightarrow$	$\neg(\bot)$	$\neg(\Gamma \subseteq \Sigma)$	$\Rightarrow$	$\neg(\boldsymbol{\Gamma} \subseteq \underline{\boldsymbol{\Sigma}})$, $\neg(\Gamma \equiv \underline{\Sigma})$
$\neg(\Gamma \supseteq \Sigma)$	$\Rightarrow$	$\neg(\boldsymbol{\Gamma} \supseteq \overline{\boldsymbol{\Sigma}})$, $\neg(\Gamma \equiv \overline{\Sigma})$	$\neg(\Gamma \supseteq \Sigma)$	$\Rightarrow$	$\neg(\bot)$
$\neg(\Gamma \cap \Sigma \neq \emptyset)$	$\Rightarrow$	$\neg(\bot)$	$\neg(\Gamma \cap \Sigma \neq \emptyset)$	$\Rightarrow$	$\neg(\boldsymbol{\Gamma} \cap \underline{\boldsymbol{\Sigma}} \neq \emptyset)$, $\neg(\Gamma \supseteq \underline{\Sigma})$
$\neg(\Gamma \cap \Sigma = \emptyset)$	$\Rightarrow$	$\neg(\Gamma \cap \overline{\Sigma} = \emptyset)$	$\neg(\Gamma \cap \Sigma = \emptyset)$	$\Rightarrow$	$\neg(\bot)$

Figure 5: Selection criteria and required pruning conditions for $\underline{\Sigma}$ and $\overline{\Sigma}$ (right side of the implications). Instead of $\mathcal{M}(\phi)$ we simply write ϕ.

If no particular relation could be deduced, we used the truth symbol $\top$ (i.e., to make the implication a tautology).

The lower table in Figure 5 applies if the selection predicate appears negated in the query formula (e.g., in the form $\neg select(...)$). For determining the pruning conditions for this case, we used the fact that $A \Rightarrow B$ is logically equivalent to $\neg B \Rightarrow \neg A$ and the previously determined implications of positive relations between Γ and $\underline{\Sigma}$ resp. $\overline{\Sigma}$.

Efficient Directory Search

The need for efficient discovery and matchmaking leads to a need for search structures and indexes for directories. We consider service descriptions represented as propositional formulas as multidimensional data and we use techniques related to the indexing of such kind of information for organizing the directory.

The indexing technique we use is based on the Generalized Search Tree (GiST) structure, which was initially proposed as a unifying framework by Hellerstein (Hellerstein, Naughton, & Pfeffer 1995) and later extended regarding aspects such as concurrency. 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 (Σ_i) that satisfy a query predicate (Γ), 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.

Relevant to our work are also SS trees, the GiST extensions described in (Aoki 1998) regarding heuristic directed stateful search. The main difference between SS trees and our approach is that we uses a declarative query language which makes the internal organization of the directory transparent to the user. In our case, search is still highly efficient thanks to a query transformation scheme that exploits the tree structure of the index.

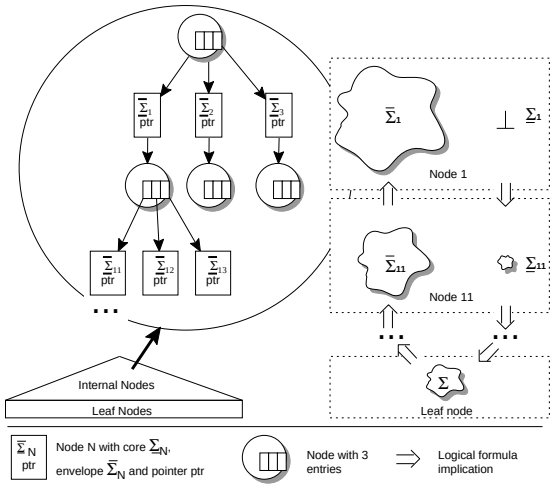

Figure 6: Theory approximation tree.

Our approach extends the basic GiST framework by associating a second predicate, which is subsumed by all values below in the tree, with each inner node. 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 ($\neg select(...)$).

As it can be seen in the example in Figure 6, while the size of envelope predicates normally increases as they are closer to the root of the tree, core predicates become smaller or even empty ($\bot$) as they approach the root.

In our implementation core formulas are constructed as the intersection of the envelopes of the core formulas in children nodes. In the example in Figure 6, for the inner node 1 with child nodes 11, 12, and 13, this is:

$$\underline{\Sigma}_1 = \underline{\Sigma}_{11} \cap \underline{\Sigma}_{12} \cap \underline{\Sigma}_{13}.$$

Conversely, envelope formulas are constructed as the union of the envelopes of the formulas below:

$$\overline{\Sigma}_1 = \overline{\Sigma}_{11} \cup \overline{\Sigma}_{12} \cup \overline{\Sigma}_{13}.$$

In our implementation we use 0-suppressed binary decision diagrams (ZDDs) (Minato 1993) to represent formulas. ZDDs are a compressed graph representations of combination sets, allowing us to efficiently manipulate formulas, to determine inclusions between models, and to count the number of

models. This is in concordance with the assumption that service directories are optimized for queries. A higher overhead for the update of entries is tolerable.

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 user-defined 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.

While the query is being processed, the visited nodes are maintained in a heap (priority queue), where the node with the most promising heuristic value comes first. 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 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 the number of retrieved results exceed a certain threshold defined by the directory service provider.

Query Transformation

In this section we give an overview of our transformation scheme that integrates the flexibility and transparency provided by the FormulaQL 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 FormulaQL expression is applied to leaf nodes of the directory tree, which correspond to concrete service advertisements.

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 an 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 FormulaQL 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 query transformation starts by putting the `select` part of the initial formula into a Negated Normal Form (NNF) by propagating negations over boolean expressions such that they appear only in front of $select()$ constructs or numeric boolean expressions (e.g., $<$, $<=$, etc.). Numeric expression directly absorb negations by inverting the comparator (e.g., $\neg <$ $\Rightarrow$ $>=$). The second phase of the transformation relaxes the query by using the appropriate bounds. Positive $select(...)$ expressions are relaxed using the rules in the upper left part of Figure 5 applied to the upper bound approximation $\overline{\Sigma}$. For negated expressions of the form $\neg select(..)$, the lower right part of the table is used. In inner nodes, the $allSA$ quantifier is relaxed to $someSA$. 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.

```
select (allSAsomeSR IN IN OVERLAP)
order by asc
    (- (countSR OUT) (count OUT OUT SUPERSET))
```

```
select (someSAsomeSR IN IN OVERLAP)
order by asc
    (- (countSR OUT) (count OUT OUT OVERLAP))
```

Table 2: Original (top) and generated (bottom) query for service composition based on forward chaining with partial type matches.

Evaluation

We evaluated our approach with a service composition planner based on forward chaining (Constantinescu, Faltings, & Binder 2004). The planner 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.

The example in Table 2 shows a selection and ranking function suited for a service composition algorithm using forward chaining with partial type matches (Constantinescu, Faltings, & Binder 2004). Our forward chaining planner requires that all inputs needed by the service are 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 types in the service). The results are sorted in ascending order according to the number of outputs that are still missing after application of the service. The code for inner nodes is generated according to the transformation scheme described in the previous section.

We evaluated our approach by carrying out tests on random service descriptions and service composition problems. 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

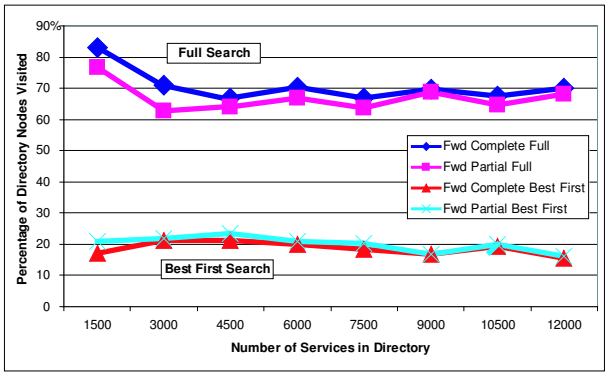

Figure 7: Average percentage of visited directory nodes per query.

matching services, too. Since we were interested in the efficiency of the directory search, we evaluated the average percentage of tree nodes visited during the processing of a query (i.e., 100% means that every node in the directory was searched) for different directory sizes.

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 results according to the provided ranking function. In the second configuration we evaluated the directory that performs a *best-first search* applying the transformed selection and ranking function to inner nodes, thus lazily creating the result set. For both directories, we used exactly the same set of service descriptions, and for each iteration, we ran the algorithms on exactly the same random composition problems.

The results (Figure 7) show that for both the complete and the partial type matches composition algorithms, the number of directory nodes that are evaluated is smaller in the case of the *best-first search* than in the case of *full search* (about 20% instead of 80%). While the directory increases in size, the percentage of visited nodes is slowly decreasing.

Conclusion

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

The directory is organized as a special kind of balanced search tree, where nodes contain also an "intersection" 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 consistently less directory nodes than a simpler directory implementation.

References

Aoki, P. M. 1998. Generalizing "search" in generalized search trees. In *Proc. 14th IEEE Conf. Data Engineering, ICDE*, 380–389. IEEE Computer Society.

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

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

Constantinescu, I., and Faltings, B. 2003. Efficient matchmaking and directory services. In *The 2003 IEEE/WIC International Conference on Web Intelligence*.

Constantinescu, I.; Faltings, B.; and Binder, W. 2004. Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS-2004)*.

Fikes, R., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. In *IJCAI*, 608–620.

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

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

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

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

Minato, S. 1993. Zero-suppressed BDDs for set manipulation in combinatorial problems. In IEEE, A.-S., ed., *Proceedings of the 30th ACM/IEEE Design Automation Conference*, 272–277. Dallas, TX: ACM Press.

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

Pednault, E. P. D. 1989. Adl: Exploringthe middle ground between strips and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, 324–332.

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

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