A Flexible Directory Query Language for the Efficient Processing of Service Composition Queries

Walter Binder[1]
Faculty of Informatics
University of Lugano
Lugano, Switzerland
walter.binder@unisi.ch

Ion Constantinescu and Boi Faltings
Ecole Polytechnique Fédérale de Lausanne (EPFL)
Artificial Intelligence Laboratory
CH-1015 Lausanne, Switzerland
{ion.constantinescu, boi.faltings}@epfl.ch

ABSTRACT:
Service composition is a major challenge in an open environment populated by large numbers of heterogeneous services. In such a setting, the efficient interaction of directory-based service discovery with service composition engines is crucial. In this article we present a Java-based directory that offers special functionality enabling effective service composition. In order to optimize the interaction of the directory with different service composition algorithms exploiting application-specific heuristics, the directory supports user-defined selection and ranking functions written in a declarative query language. Inside the directory queries are transformed and compiled to JVM bytecode which is dynamically linked into the directory. The compiled query enables a best-first search of matching directory entries, efficiently pruning the search space.[2]

KEY WORDS:
Service directories, service discovery and composition, indexing techniques, efficient query processing

1. Introduction

There is a good body of work which addresses the service composition problem with planning techniques based either on theorem proving (e.g., Golog (McIlraith, & Son, 2002), SWORD (Ponnekanti, & Fox, 2002)) or on hierarchical task planning (e.g., SHOP-2 (Wu, Parsia, Sirin, Hendler, & Nau, 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.

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

[1] This work was conducted while the first author was affiliated with the Artificial Intelligence Laboratory of the Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland.

[2]The work presented in this article was partly carried out in the framework of the EPFL Center for Global Computing and supported by the Swiss National Funding Agency OFES as part of the European projects KnowledgeWeb (FP6-507482) and DIP (FP6-507483).

adapted to a situation where planning operators are not known a priori, but have to be retrieved through queries to these directories. (Lassila, & Dixit, 2004) addressed the problem of interleaving discovery and composition, but they considered only simple workflows where services had one input and one output.

Our approach to automated service composition is based on matching input and output parameters of services using type information in order to constrain the ways how services may be composed. Our composition algorithm allows for partially matching types and handles them by introducing switches in the composition plan. Experimental results show that using partial matches significantly decreases the failure rate compared with a composition algorithm that supports only complete matches (Constantinescu, Faltings, & Binder, 2004d).

We have developed a directory service with specific features to ease service composition. Queries may not only search for complete matches, but may also retrieve partially matching directory entries (Constantinescu, & Faltings, 2003). As the number of (partially) matching entries may be large, the directory supports incremental retrieval of the results of a query. This is achieved through sessions, during which a client issues queries and retrieves the results in chunks of limited size (Constantinescu, Binder, & Faltings, 2004a).

As in a large-scale directory the number of (partially) matching results for a query may be very high, it is crucial to order the result set within the directory according to heuristics and to transfer first the better matches to the client. If the heuristics work well, only a small part of the possibly large result set has to be transferred, thus saving network bandwidth and boosting the performance of a directory client that executes a service composition algorithm (the results are returned incrementally, once a result fulfills the client's requirements, no further results need to be transmitted). However, the heuristics depend on the concrete composition algorithm. For each service composition algorithm (e.g., forward chaining, backward chaining, etc.), a different heuristic may be better adapted. Because research on service composition is still in the beginning and the directory cannot anticipate the needs of all possible service composition algorithms, our directory supports user-defined selection and ranking heuristics expressed in a declarative query language. The support for application-specific heuristics significantly increases the flexibility of our directory, as the client is able to tailor the processing of directory queries. For efficient execution, the queries are dynamically transformed and compiled by the directory.

As the main contributions of this article, we show how our directory supports user-defined selection and ranking heuristics. We present a dedicated query language and explain how queries are processed by the directory. In a first step, the directory transforms queries in order to better exploit the internal directory organization during the search. This allows a best-first search that generates the result set in a lazy way, reducing response time and workload within the directory. In a second step, the query is compiled in order to speed up the directory search. Compared with previous work (Binder, Constantinescu, & Faltings, 2004; Constantinescu, Binder, & Faltings, 2004b; Constantinescu, Binder, & Faltings, 2005), the novel, original contributions of this article are the declarative directory query language, the transformation mechanism to make better use of the internal directory structure, and the compilation of queries to JVM bytecode, which is dynamically linked into the directory. These techniques, which have not been applied in the context of service directories before, provide a flexible and efficient mechanism for query processing.

This article is structured as follows: Section 2 reviews some related work concerning service discovery and composition. Section 3 gives an overview of our service description formalism and of the internal index structure of our directory. In Section 4 we present a simple, functional query

language which allows to express application-specific selection and ranking heuristics. Section 5 explains the processing of directory queries and introduces query transformations that enable a best-first search with early pruning. In Section 6 we explain the dynamic compilation of queries. Section 7 gives a brief overview of our service composition algorithms. Section 8 discusses some sample queries and shows how they are compiled by the directory. In Section 9 we present experimental results, which confirm that our novel way of query processing significantly improves the efficiency of directory search. Finally, Section 10 concludes this article.

2. Related Work

In this section we review some related work regarding service discovery and service composition.

Service Discovery

UDDI has become the de-facto standard as a general framework to describe and discover services and web service providers. More specifically, reference (UDDI) is a technical note that details how WSDL descriptions of web services can be mapped to UDDI data structures and provides examples of how to find web service descriptions using the standard UDDI query interface.

Within the academic world, a number of approaches exist that try to build semantically enhanced discovery components on top of UDDI. (Kawamura, Blasio, Hasegawa, Paolucci, & Sycara, 2003) augment the standard UDDI registry APIs with semantic annotations. (Verma, Sivashanmugam, Sheth, Patil, Oundhakar, & Miller, 2004) use a set of distributed UDDI registries as storage layer, where each registry is mapped to a specific domain based on a registry ontology.

A very active research field is the development of discovery algorithms. There, the main focus is on finding good ‘approximations’ when a perfect match cannot be found, i.e., the advertised capabilities of a service do not fully match the request. (Paolucci, Kawamura, Payne, & Sycara, 2002) define an ordered scoring function based on input and output parameters.

A similar approach is taken in (Li, & Horrocks, 2003) and applied to concept-based reasoning in description logics (Baader, & Sattler, 2001). Here, the above categorization is extended by an intersection match, where the intersection of the concepts of the request and the concepts of the advertisement is satisfiable. This intersection match is ranked below a subsumption match.

(Benatallah, Hacid, Rey, & Toumani, 2003) describe an interesting approach where discovery is treated as a query rewriting problem on hypergraphs: It attempts to find the so-called best profile cover which is defined as the set of web services that satisfies as many outputs of the request as possible, while requiring as few non-requested inputs as possible.

While the above approaches to discovery allow for a few different match types to be supported by the directory, the query formalisms are quite coarse and in particular lack support for a good integration of the discovery process with service composition. In contrast, our directory offers a much more flexible query interface, yet ensures efficient query processing with the aid of transformation and compilation techniques.

Service Composition

Some approaches to service composition require an explicit specification of the control flow between basic services in order to provide value-added services. For instance, in the eFlow system (Casati, Ilnicki, Jin, Krishnamoorthy, & Shan, 2000), a composite service is modeled as a graph that defines the order of execution of different processes. The Self-Serv framework (Benatallah, Sheng, & Dumas, 2003) uses a subset of statecharts to describe the control flow within a composite service. The Business Process Execution Language for Web Services (BPEL4WS) addresses compositions where the control flow of the process and the bindings between services are known in advance.

The most prominent body of work regarding the automatic composition of services concerns approaches that address the composition problem with planning techniques, based either on theorem proving (e.g., ConGolog (McIlraith, Son, & Zeng, 2001; McIlraith, & Son, 2002), SWORD (Ponnekanti, & Fox, 2002)) or on hierarchical task planning (e.g., SHOP-2 (Wu, Parsia, Sirin, Hendler, & Nau, 2003)).

Other planning techniques, such as planning as model checking (Giunchiglia, & Traverso, 1999), are being considered for web service composition and allow more complex constructs such as loops. While our approach aims at 'functional-level' service composition by selecting and combining services to match given user requirements, 'process-level' service composition (Traverso, & Pistore, 2004) is able to handle complex service interactions. These two approaches are complementary: Integrating 'process-level' service composition after an initial 'functional-level' service composition step would allow to deal with complex service protocols.

All these planning-based approaches assume that available service descriptions are initially loaded into a reasoning engine and that no discovery is performed during composition. However, in an open environment populated by a large number of dynamically changing services, service advertisements are usually stored in external, possibly large-scale directories. Hence, it is not practical to copy the whole contents of a remote directory into the local reasoning engine before starting the composition algorithm.

In contrast to the aforementioned approaches, our research focuses on the efficient, dynamic interaction of service composition algorithms with an external directory in order to incrementally retrieve relevant service descriptions during the composition.

3. Service Descriptions and Directory Index

Service descriptions are a key element for service discovery and service composition, as they enable automated interactions between applications. In our system, a service is described by two parameter sets – the *input parameters required by the service* and the *output parameters provided by the service*. Each parameter is identified by a unique name in its set and has an associated parameter type, which specifies the space of possible values for the parameter. We require that parameter types are not empty, i.e., there must be at least one allowed value for each parameter.

Parameter types can be either sets of intervals of basic data types (e.g., date/time, integers, floating-point numbers) or classes, which may be defined in a descriptive language such as OWL. From the class descriptions we derive a directed graph of simple is-a relations by using a description logic classifier. Details concerning the encoding of parameters are presented in (Constantinescu, & Faltings, 2003).

Despite its simplicity, our service description formalism is consistent with existing formalisms, such as WSDL and OWL-S. The input/output specification of services described in WSDL or OWL-S can be mapped to our simplified formalism.

The need for efficient discovery and matchmaking leads to a need for search structures and indexes for directories. We consider service descriptions as multidimensional data and use techniques related to the indexing of such kind of information in the directory. Our directory index is based on the *Generalized Search Tree (GiST)*, proposed as a unifying framework by (Hellerstein, Naughton, & Pfeffer, 1995). The design principle of the 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 index and the leaf nodes point to the actual data.

Figure 1 presents a schematic view of the general GiST structure. Each leaf node in the GiST of our directory holds references to all service descriptions with a certain input/output characterization. The required inputs of the service and the provided outputs (sets of parameter names with associated types) are stored in the leaf node. For inner nodes of the tree, an *upper bound* (sometimes called 'envelope' in the literature) of all input/output parameters found in the subtree is stored. Conceptually, each inner node I on the path to a leaf node L contains all input/output parameters stored in L. The type associated with a parameter in I subsumes the type of the parameter in L. I.e., for an inner node, the input/output parameters indicate which concrete parameters may be found in a leaf node of the subtree. If a parameter is not present in an inner node, it will not be present in any leaf node of the subtree.

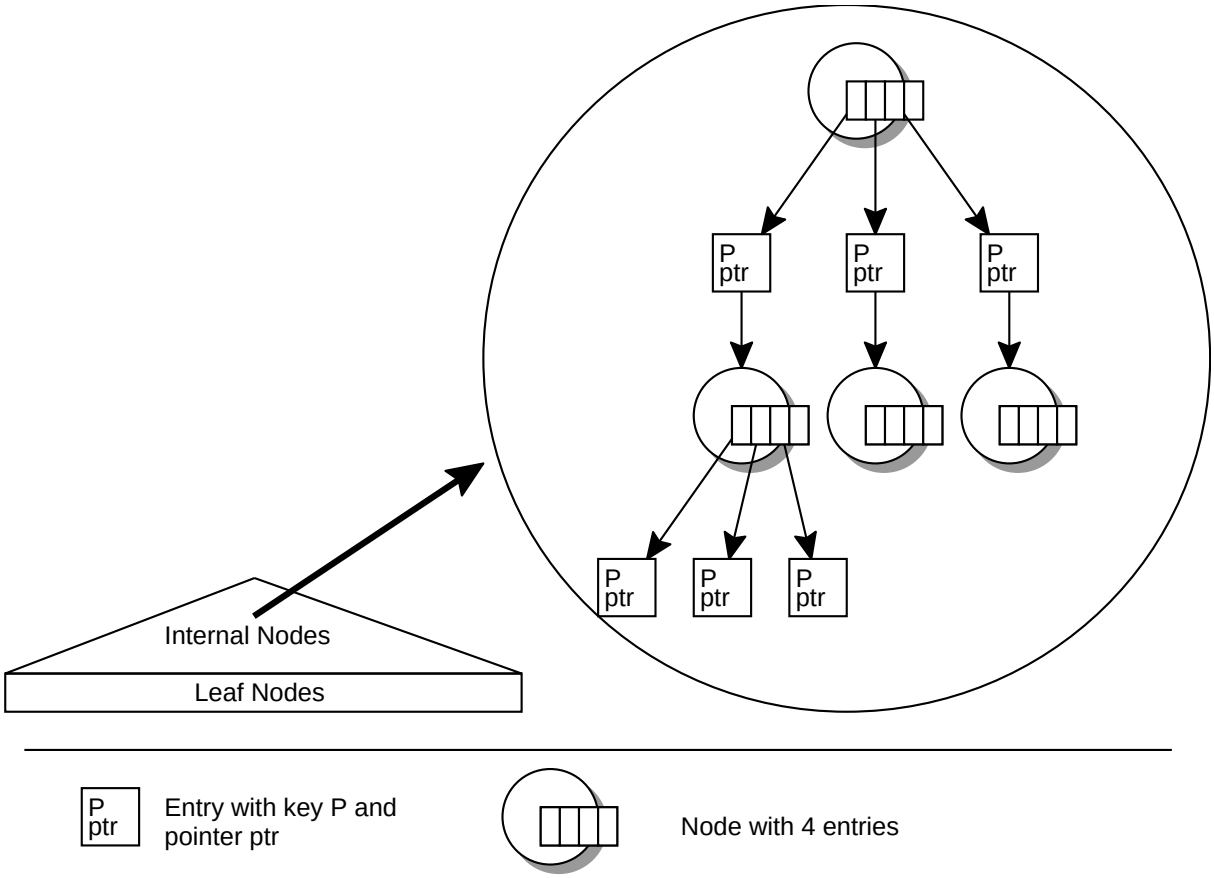

Figure 1. Schematic view of the Generalized Search Tree (GiST).

An extension of the GiST, where inner nodes also store a lower bound (sometimes called 'core') in order to improve pruning for certain types of queries, was presented in (Constantinescu, Binder, & Faltings, 2005).

4. Directory Query Language

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 (Constantinescu, Binder, & Faltings, 2004a). 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 composition 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 composition 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. This approach can be seen as a form of remote evaluation (Fuggetta, Picco, & Vigna, 1998).

A directory query consists of a set of provided inputs and required outputs (both sets contain tuples of parameter name and associated type), as well as a custom selection and ranking function. The selection and ranking function is written in the simple, high-level, functional query language $DirQL_{SE}$ (Directory Query Language with Set Expressions). An (informal) EBNF grammar for $DirQL_{SE}$ is given in Table 1. The non-terminal *constant*, which is not shown in the grammar, represents a non-negative numeric constant (integer or decimal number). The syntax of $DirQL_{SE}$ has some similarities with LISP.[3] We have designed the language considering the following requirements:

- **Simplicity:** $DirQL_{SE}$ offers only a minimal set of constructs, but it is expressive enough to write relevant selection and ranking heuristics.
- **Declarative:** $DirQL_{SE}$ is a functional language and does not support destructive assignment. The absence of side-effects eases program analysis and transformations.
- **Safety:** As the directory executes user-defined code, $DirQL_{SE}$ expressions must not interfere with internals of the directory. Moreover, the resource consumption (e.g., CPU, memory) needed for the execution of $DirQL_{SE}$ expressions is bounded in order to prevent denial-of-service attacks: $DirQL_{SE}$ supports neither recursion nor loops, and queries can be executed without dynamic memory allocation.
- **Efficient directory search:** $DirQL_{SE}$ has been designed to enable an efficient best-first search in the directory GiST. Code transformations automatically generate selection and ranking functions for the inner nodes of the GiST (see Section 5).
- **Efficient compilation:** Due to the simplicity of the language, $DirQL_{SE}$ expressions can be efficiently compiled to increase performance (see Section 6).

[3] In order to simplify the presentation, in this article the operators '`and`', '`or`', '`<`', '`>`', '`<=`', '`>=`', '`=`', '`+`', '`*`', '`-`', '`min`', and '`max`' are binary, whereas in the implementation they may take an arbitrary number arguments, similar to the definition of these operations in LISP.

```
dirqlExpr : selectExpr
          | rankExpr
          | selectExpr rankExpr

selectExpr : 'select' booleanExpr

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

booleanExpr: '(' ('and' | 'or') booleanExpr booleanExpr ')'
           | '(' 'not' booleanExpr ')'
           | '(' ('<'|'>'|'<='|'>='|'=') numExpr numExpr ')'

numExpr : constant
        | '(' ('+' | '*' | '-' | '/') numExpr numExpr ')'
        | '(' ('min' | 'max') numExpr numExpr ')'
        | '(' 'if' booleanExpr numExpr numExpr ')'
        | setExpr

setExpr : '(' 'union' querySet serviceSet ')'
        | '(' 'intersection' querySet serviceSet typeTest ')'
        | '(' 'minus' querySet serviceSet typeTest ')'
        | '(' 'minus' serviceSet querySet typeTest ')'
        | '(' 'size' (querySet | serviceSet) ')'

querySet : ('qin' | 'qout')

serviceSet : ('sin' | 'sout')

typeTest : ('FALSE' | 'EQUAL' | 'S_CONTAINS_Q' |
            'Q_CONTAINS_S' | 'OVERLAP' | 'TRUE')
```

Table 1. A grammar for $DirQL_{SE}$.

A $DirQL_{SE}$ expression defines custom selection and ranking heuristics. The evaluation of a $DirQL_{SE}$ expression is based on the 4 sets `qin` (available inputs specified in the query), `qout` (required outputs specified in the query), `sin` (required inputs of a certain service S), and `sout` (provided outputs of a certain service S). Each element in each of these sets represents a query/service parameter identified by its unique name within the set and has an associated type.

A $DirQL_{SE}$ expression may involve some simple arithmetic. The result of a numeric $DirQL_{SE}$ expression is always non-negative. The '`-`' operator returns 0 if the second argument is greater than the first one. The $DirQL_{SE}$ programmer may use the '`if`' conditional to ensure that the first argument of '`-`' is greater than or equal to the second argument. For division, the second operand (divisor) has to evaluate to a constant for a given query. That is, the divisor is a numeric expression built from constant subexpression; as `qin` and `qout` are constant for a query, the divisor may include `size(qin)` and `size(qout)` as subexpressions. Before a query is

executed, the directory ensures that the $DirQL_{SE}$ expression will not cause a division by zero. For this purpose, all subexpressions are examined. The reason for these restrictions will be explained in the following section.

A $DirQL_{SE}$ query may comprise a selection and a ranking expression. Service descriptions (inputs/outputs defined by `sin`/`sout`) for which the selection expression evaluates to *false* are not returned to the client (pruning). The ranking expression defines the custom ranking heuristics. For a certain service description, the ranking expression computes a non-negative value. The directory will return service descriptions in ascending or descending order, as specified by the ranking expression.

The selection and ranking expressions may make use of several set operations. `size` returns the cardinality of any of the sets `qin`, `qout`, `sin`, or `sout`. The operations `union`, `intersection`, and `minus` take as arguments a query set (`qin` or `qout`) as well as a service set (`sin` or `sout`). For `union` and `intersection`, the query set has to be provided as the first argument. All set operations return the cardinality of the resulting set.

- **union:** Cardinality of the union of the argument sets. Type information is irrelevant for this operation.
- **intersection:** Cardinality of the intersection of the argument sets. For a parameter to be counted in the result, it has to have the same name in both argument sets and the type test (third argument) has to succeed.
- **minus:** Cardinality of the set minus of the argument sets (first argument set minus second argument set). For a parameter to be counted in the result, it has to occur in the first argument set and, either there is no parameter with the same name in the second set, or in the case of parameters with the same name, the type test has to fail.

The type of parameters cannot be directly accessed, only the operations `intersection` and `minus` make use of the type information. For these operations, a type test is applied to parameters that have the same name in the given query and service set. The following type tests are supported (T_S denotes the type of a common parameter in the service set, while T_Q is the type of the parameter in the query set): `FALSE` (always fails), `EQUAL` (succeeds if $T_S = T_Q$), `S_CONTAINS_Q` (succeeds if T_S subsumes T_Q), `Q_CONTAINS_S` (succeeds if T_Q subsumes T_S), `OVERLAP` (succeeds if there is an overlap between T_S and T_Q, i.e., if a common subtype of T_S and T_Q exists), and `TRUE` (always succeeds).

```
Services:

S1: sin  = { from: Airport }
    sout = { destinations: AirportList }

S2: sin  = { from: Airport, to: Airport }
    sout = { airline: String }

S3: sin  = { from: Airport, to: Airport }
    sout = { airline: String, price: Currency }



Query Q:

qin  = { from: Airport, to: Airport }
qout = { airline: String, price: Currency }

select (and (<= (size qin) (size sin))
            (<= (size sin) (intersection qin sin EQUAL)))
order by desc (intersection qout sout TRUE)
```

Table 2. Example services and query.

A simple example is shown in Table 2. The directory maintains three services, $S1$, $S2$, and $S3$. $S1$ requires one input `from` (type `Airport`) and provides as single output `destinations` (type `AirportList`) a list of directly reachable airports. $S2$ requires two inputs `from` and `to` (both of type `Airport`) and provides as single output `airline` (type `String`) the name of an airline offering flights between the two airports. $S3$ resembles $S2$, but in addition to `airline`, $S3$ produces also a second output `price` (type `Currency`) representing the price of a ticket for the desired flight.

The query Q provides `qin` and `qout`, as well as a $DirQL_{SE}$ expression. The selection expression selects those services for which `sin` exactly matches the given `qin`. As $DirQL_{SE}$ does not support the direct comparison of sets, we require the cardinalities of `qin`, `sin`, and of the intersection of `qin` and `sin` (exact type matches) to be equal. The selection expression shown in Table 2 has been obtained by taking the following equivalence for sets A and B into consideration: $A = B \Leftrightarrow |A| \leq |B| \leq |A \cap B|$. Note that this selection expression does not impose any restrictions on the service outputs. Services $S2$ and $S3$, but not service $S1$, match this selection expression.

The ranking expression shown in Table 2 sorts the services according to the overlap of `sout` with `qout` in descending order. The ranking expression does not take types into account. In this example, the directory will return $S3$ before $S2$, because the outputs of $S3$ have a higher overlap with `qout` than the outputs of $S2$.

Examples of more useful $DirQL_{SE}$ queries will be shown in Section 7 and Section 8.

5. Efficient Directory Search

Processing a user query requires traversing the GiST structure of the directory starting from the root node. The given $DirQL_{SE}$ expression is applied to leaf nodes of the directory tree, which correspond to concrete service descriptions (i.e., `sin` and `sout` represent the exact input/output parameters of a service description). For an inner node I of the GiST, `sin` and `sout` are supersets of the input/output parameters found in any node of the subtree whose root is I. The type of each parameter in I is a supertype of the parameter found in any node (which has a parameter with the same name) in the subtree. Therefore, the user-defined selection and ranking function cannot be directly applied to inner nodes.

In order to prune the search (as close as possible to the root of the GiST) and to implement a best-first search strategy which expands the most promising branch in the tree first, appropriate selection (pruning) and ranking functions are needed for the inner nodes of the GiST. In our approach, 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 the 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). Figure 2 illustrates the processing of a directory query.

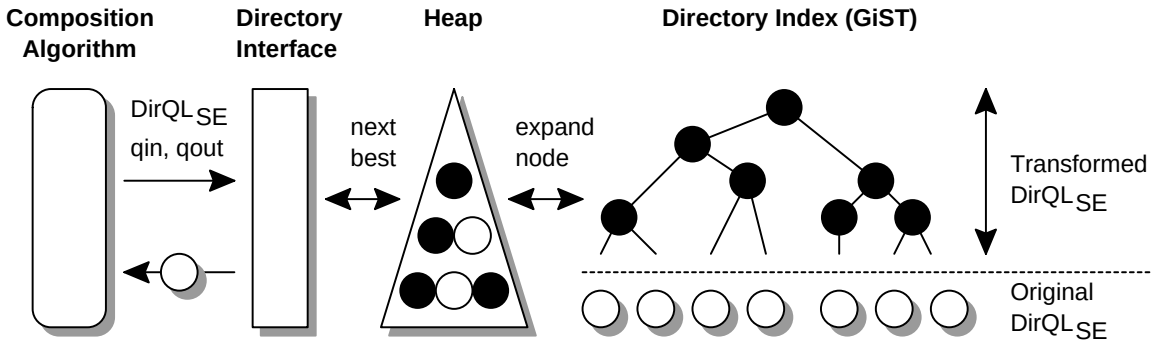

Figure 2. Processing of a directory query. While the given $DirQL_{SE}$ expression is directly applied to leaf nodes (white), it has to be transformed for inner nodes (black).

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. 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. 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.

Table 3 shows the transformation operators $\uparrow$ and $\downarrow$ which allow to generate the code for calculating upper and lower bounds in inner nodes of the GiST. The variables a and b are arbitrary numeric expressions, c is a numeric expression that is guaranteed to be constant throughout a query, x is a boolean expression, q may be `qin` or `qout`, s may be `sin` or `sout`, and t is a type test. The operator $\oplus$ relaxes certain type tests, the operator $\ominus$ constrains them. For a $DirQL_{SE}$ ranking expression '`order by asc` E', the code for inner node ranking is '`order by asc` $\downarrow E$'; for a ranking expression '`order by desc` E', the inner node ranking code is '`order by desc` $\uparrow E$'.

$\uparrow constant$	$\rightarrow$	$constant$	$\downarrow constant$	$\rightarrow$	$constant$
$\uparrow(+\ a\ b)$	$\rightarrow$	$(+\uparrow a\uparrow b)$	$\downarrow(+\ a\ b)$	$\rightarrow$	$(+\downarrow a\downarrow b)$
$\uparrow(*\ a\ b)$	$\rightarrow$	$(*\uparrow a\uparrow b)$	$\downarrow(*\ a\ b)$	$\rightarrow$	$(*\downarrow a\downarrow b)$
$\uparrow(-\ a\ b)$	$\rightarrow$	$(-\uparrow a\downarrow b)$	$\downarrow(-\ a\ b)$	$\rightarrow$	$(-\downarrow a\uparrow b)$
$\uparrow(/\ a\ c)$	$\rightarrow$	$(/\uparrow a\ c)$	$\downarrow(/\ a\ c)$	$\rightarrow$	$(/\downarrow a\ c)$
$\uparrow(min\ a\ b)$	$\rightarrow$	$(min\uparrow a\uparrow b)$	$\downarrow(min\ a\ b)$	$\rightarrow$	$(min\downarrow a\downarrow b)$
$\uparrow(max\ a\ b)$	$\rightarrow$	$(max\uparrow a\uparrow b)$	$\downarrow(max\ a\ b)$	$\rightarrow$	$(max\downarrow a\downarrow b)$
$\uparrow(if\ \ x\ a\ b)$	$\rightarrow$	$(max\uparrow a\uparrow b)$	$\downarrow(if\ \ x\ a\ b)$	$\rightarrow$	$(min\downarrow a\downarrow b)$
$\uparrow(union\ q\ s)$	$\rightarrow$	$(union\ q\ s)$	$\downarrow(union\ q\ s)$	$\rightarrow$	$(size\ q)$
$\uparrow(intersection\ q\ s\ t)$	$\rightarrow$	$(intersection\ q\ s\oplus t)$	$\downarrow(intersection\ q\ s\ t)$	$\rightarrow$	0
$\uparrow(minus\ q\ s\ t)$	$\rightarrow$	$(size\ q)$	$\downarrow(minus\ q\ s\ t)$	$\rightarrow$	$(minus\ q\ s\oplus t)$
$\uparrow(minus\ s\ q\ t)$	$\rightarrow$	$(minus\ s\ q\ominus t)$	$\downarrow(minus\ s\ q\ t)$	$\rightarrow$	0
$\uparrow(size\ q)$	$\rightarrow$	$(size\ q)$	$\downarrow(size\ q)$	$\rightarrow$	$(size\ q)$
$\uparrow(size\ s)$	$\rightarrow$	$(size\ s)$	$\downarrow(size\ s)$	$\rightarrow$	0
$\oplus TRUE$	$\rightarrow$	$TRUE$	$\ominus TRUE$	$\rightarrow$	$TRUE$
$\oplus OVERLAP$	$\rightarrow$	$OVERLAP$	$\ominus OVERLAP$	$\rightarrow$	$FALSE$
$\oplus Q\_CONTAINS\_S$	$\rightarrow$	$OVERLAP$	$\ominus Q\_CONTAINS\_S$	$\rightarrow$	$Q\_CONTAINS\_S$
$\oplus S\_CONTAINS\_Q$	$\rightarrow$	$S\_CONTAINS\_Q$	$\ominus S\_CONTAINS\_Q$	$\rightarrow$	$FALSE$
$\oplus EQUAL$	$\rightarrow$	$S\_CONTAINS\_Q$	$\ominus EQUAL$	$\rightarrow$	$FALSE$
$\oplus FALSE$	$\rightarrow$	$FALSE$	$\ominus FALSE$	$\rightarrow$	$FALSE$

Table 3. Transformation operators $\uparrow$, $\downarrow$, $\oplus$, and $\ominus$ for the generation of inner node code.

If I is an inner node on the path to the leaf node L and E is a $DirQL_{SE}$ ranking expression, $\uparrow E$ (resp. $\downarrow E$) applied to I has to compute an upper (resp. lower) bound for E applied to L. In the following we exemplarily explain 3 transformation rules of Table 3; the remaining rules can be explained in a similar way.

First we consider computing an upper bound for $E = (intersection\ q\ s\ t)$. In an inner node I, the service set s_I is a superset of s_L in a leaf node, while the query set q remains constant. Moreover, the type of each parameter in s_L is subsumed by the type of the parameter with the same name in s_I. Not considering the parameter types, applying E to I would compute an upper bound for E applied to L, as intuitively the intersection of q with the bigger set s_I will not be smaller than the intersection of q with s_L. Taking parameter types into consideration, we must ensure that whenever a type test succeeds for L, it will also succeed for I. That is, if a common parameter is counted in the intersection in L, it must be also counted in the intersection in I. As it can be seen in Table 3, $\oplus t$ will succeed in I, if t succeeds in L (remember that parameter types are guaranteed to be non-empty). For instance, if the type of a parameter in s_L is subsumed by the type of the parameter with the same name in q (`Q_CONTAINS_S` succeeds for that parameter in L), the type of the corresponding parameter in s_I (which subsumes the type in s_L) will overlap with the parameter type in q. If the types in s_L and q are equal, the type in s_I will subsume the type in q.

As a second example we want to compute an upper bound for $E = (minus\ s\ q\ t)$. Without considering parameter types, applying E to I would give an upper bound for E applied to L, as s_I is a superset of s_L. In contrast to the `intersection` operation, a common parameter is counted in the result if the type test fails. That is, if the type test fails in L, it has also to fail in I. As shown in table Table 3, $\ominus t$ will fail in I, if t fails in L. For example, if the type of a parameter in q does not subsume the type of the parameter with the same name in s_L (`Q_CONTAINS_S` fails for that parameter in L), it will also not subsume the type of that parameter in s_I (which subsumes the type of the parameter in s_L). If the type test is `TRUE`, it will never fail, neither in L nor in I. In all other cases, no matter whether the type test fails in L or not, it will fail in I (because $\ominus t$ will be `FALSE`). Hence, '$\uparrow$ `(minus s q t)`' may result in '`(minus s q FALSE)`', which is equivalent to '`(size s)`'.

As a last example, consider the calculation of an upper bound for $(min\ a\ b)$. The transformation rule defines $\uparrow(min\ a\ b) = (min\ \uparrow a\ \uparrow b)$. We assume that $\uparrow a \geq a$ and $\uparrow b \geq b$. We have to show that in all possible cases $\uparrow(min\ a\ b) \geq (min\ a\ b)$:

- $(min\ a\ b) = a \wedge (min\ \uparrow a\ \uparrow b) = \uparrow a \rightarrow \uparrow(min\ a\ b) = \uparrow a \geq a = (min\ a\ b)$
- $(min\ a\ b) = b \wedge (min\ \uparrow a\ \uparrow b) = \uparrow a \rightarrow \uparrow(min\ a\ b) = \uparrow a \geq a \geq b = (min\ a\ b)$
- $(min\ a\ b) = a \wedge (min\ \uparrow a\ \uparrow b) = \uparrow b \rightarrow \uparrow(min\ a\ b) = \uparrow b \geq b \geq a = (min\ a\ b)$

- $(min\ a\ b) = b \wedge (min \uparrow a \uparrow b) = \uparrow b \rightarrow \uparrow(min\ a\ b) = \uparrow b \geq b = (min\ a\ b)$

Considering the upper bound operator $\uparrow$, the reason why we require the divisor of '`/`' to evaluate to a constant becomes apparent: If c was not constant, for division the operator $\uparrow$ would have been defined as '$\uparrow(/\ a\ c) \rightarrow (/ \uparrow a \downarrow c)$'. Hence, even if the ranking expression provided by the client did not divide by zero ($c > 0$), the automatically generated code for computing an upper bound in inner nodes might possibly result in a division by zero ($\downarrow c = 0$). This would result in a runtime exception not expected by the client. For this reason, c must depend neither on `sin` nor on `sout`.

In order to automatically generate the code for inner node selection (pruning), we define the transformation operator $\updownarrow$ for boolean expressions (see Table 4). If E is *true* for a leaf node L, $\updownarrow E$ has to be *true* for all nodes on the path to L. In other words, if $\updownarrow E$ is *false* for an inner node, it must be guaranteed that E will be *false* for each leaf in the subtree. This condition ensures that during the search an inner node may be discarded (pruning) only if it is sure that all leaves in the subtree are to be discarded, too. For a $DirQL_{SE}$ selection expression '`select` E', the code for inner node selection is '`select` $\updownarrow E$'. In Table 4 a and b are numeric expressions, while x and y are boolean expressions.

$\updownarrow(and\ x\ y) \rightarrow (and \updownarrow x \updownarrow y)$	$\updownarrow(or\ x\ y) \rightarrow (or \updownarrow x \updownarrow y)$
$\updownarrow(<\ a\ b) \rightarrow (< \downarrow a \uparrow b)$	$\updownarrow(<=\ a\ b) \rightarrow (<= \downarrow a \uparrow b)$
$\updownarrow(>\ a\ b) \rightarrow (> \uparrow a \downarrow b)$	$\updownarrow(>=\ a\ b) \rightarrow (>= \uparrow a \downarrow b)$

Table 4. Transformation operator $\updownarrow$ for the generation of code in inner nodes of the GiST.

The alert reader may have noticed that the operators '`not`' and '`=`' have been omitted in Table 4. The reason for this omission is that initially we transform all boolean expressions in the query according to De Morgan's theorem, moving negations towards the leaves, removing double negations, and changing the comparators if needed. For example, the boolean expression `(not (and (>= a b) (not (and (< c 3) (> d 0)))))` is transformed to the equivalent expression `(or (< a b) (and (< c 3) (> d 0)))`. The resulting expressions are free of negations. Moreover, an expression of the form `(= a b)` is transformed to the equivalent expression `(and (<= a b) (<= b a))`.

Related to our work are SS trees (Aoki, 1998), a GiST extension for directed stateful search. The main difference between SS trees and our approach is that we use a declarative query language which makes the internal organization of the directory transparent to the user. In our system, search is still very efficient thanks to query transformation and compilation.

6. Efficient Query Execution

As the custom selection and ranking functions may be invoked very often, interpretation would cause high overhead. Thus, the directory includes a fast compiler for $DirQL_{SE}$ expressions. Because our extensible directory is entirely programmed in Java, the $DirQL_{SE}$ compiler directly generates JVM bytecode which is linked into the same JVM that executes the core functionality of the directory.

Compiling and integrating user-defined code into the directory leverages state-of-the-art optimizations in recent JVM implementations. For instance, the HotSpot VM first interprets JVM bytecode and gathers execution statistics. If code is executed frequently enough, it is compiled to optimized native code for fast execution. In this way, frequently used selection and ranking functions are executed as efficiently as algorithms directly built into the directory.

A custom selection and ranking function written in $DirQL_{SE}$ is compiled to a Java class implementing the `Ranking` interface defined in Table 5. The user-defined selection and ranking expressions are compiled into the single method `rankLeaf()`. If a selection expression is defined and evaluates to *false* , the `rankLeaf()` and `rankInner()` methods return a negative value to indicate that the current search path is to be pruned. (Remember that the ranking expression is always non-negative.) This compilation scheme has the advantage that common set expressions, which appear in the selection as well as in the ranking expression, may be evaluated only once and stored in a local variable of the method. The compiled code is structured as illustrated in Table 6. The method `rankInner()` is automatically generated by applying the transformation rules described in Section 5.

```
public interface Ranking {
   double rankLeaf( ParamSet qin, ParamSet qout,
                    ParamSet sin, ParamSet sout );
   double rankInner(ParamSet qin, ParamSet qout,
                    ParamSet sin, ParamSet sout );
}

public interface ParamSet {
   static final int FALSE=1, EQUAL=2, S_CONTAINS_Q=3,
                    Q_CONTAINS_S=4, OVERLAP=5, TRUE=6;

   int union( ParamSet p );
   int intersection( ParamSet p, int typeTest );
   int minus( ParamSet p, int typeTest );
   int size();
}
```

Table 5. The Java API for compiled queries.

```
if (!selectExpr) // omitted if there is no selectExpr
   return -1.0d;
return (double)rankExpr; // omitted if there is no rankExpr
return 0.0d; // default; omitted if there is a rankExpr
```

Table 6. Structure of compiled `rankLeaf()` and `rankInner()` methods.

The $DirQL_{SE}$ compiler involves several (logical) passes as listed below.[4] Because of the simplicity of $DirQL_{SE}$ expressions, the compilation is very fast. Each of these passes requires execution time linear with the size of the $DirQL_{SE}$ expression. Only simple and computationally cheap optimizations are performed, as the just-in-time compiler of a modern JVM will apply more sophisticated optimizations when it pays off.

1. Parsing of the given $DirQL_{SE}$ expression. Thanks to the prefix notation used in $DirQL_{SE}$, a simple recursive top-down parser can be used. The parsing pass results in an object representation of the abstract syntax tree (AST).
2. Verification. Concerning division, $DirQL_{SE}$ imposes constraints on the divisor (it must depend neither on `sin` nor on `sout`). This restriction is verified and the query is rejected in case of violation.
3. Removing negations. The AST is transformed in order to remove all negations, as explained in Section 5.
4. Generation of the selection and ranking code for the inner nodes of the GiST according to the transformation rules explained in Section 5. This pass processes the AST produced by the previous pass and generates a second AST for the inner node code. Before the transformation operators $\updownarrow$, $\uparrow$, and $\downarrow$ are applied, all occurrences of '`=`' are removed, as mentioned in Section 5.
5. Optimizing the inner node code. Simple algebraic transformations are applied to the AST of the inner node code. Due to the previous transformations, some subexpressions may be simplified or computed statically.
6. Finding common set expressions in the AST. For each of the two ASTs, common set expressions (used in the selection and in the ranking part) are identified and assigned to local variables (which are not used to pass method arguments).
7. Computing the data types of expressions. Constants in $DirQL_{SE}$ may be integers or decimal numbers. With the exception of division, integer instructions are used as long as all operands in an arithmetic expression are integers. However, if an operand is a decimal number (`double`), all other operands have to be converted to `double`. For division, both operands are always converted to `double`. If the whole ranking expression would generate an integer value, the result is finally cast to a `double` value.
8. Code generation for both ASTs, resulting in a class that implements the `Ranking` interface.

Generating first Java code and then using the standard Java compiler to generate bytecode would cause too much overhead. Hence, the last pass (code generation) directly generates JVM bytecode

[4]For efficiency reasons, the actual implementation interleaves some of these passes.

using BCEL (Dahm, 1999).[5] As the JVM is a stack machine, the code generation is very easy. The code generator is defined as a set of recursive methods that traverse the AST. For each expression, first the code of its subexpressions is generated (leaving the evaluated subexpressions on the stack), then the code for the main function is appended which consumes the values of the subexpressions and pushes the result onto the stack. Common set expressions (as identified by the sixth compilation pass) are treated differently, they are stored in their assigned local variable and loaded onto the stack on demand. For the computation of `min` (resp. `max`), a local variables is used to temporarily store the lowest (resp. highest) value of subexpressions evaluated so far. For the compilation of `min`, `max`, and `if`, branch and jump instructions are used. Because $DirQL_{SE}$ expressions have no side-effects, 'jumping' code is generated for `and` and `or` expressions (boolean short evaluation). The '`-`' operator is treated specially as well, a conditional is inserted to replace the result with 0 if it would be negative otherwise.

Due to the simplicity of $DirQL_{SE}$, the execution time of a custom selection and ranking function is bounded by the size of the expression. Hence, an attacker cannot crash the directory by providing, for example, a query that contains an endless loop. Moreover, compiled queries do not allocate any memory during their execution. The set operations return the cardinality of sets, as the allocation of new set objects would cause significantly more overhead. As a prevention against denial-of-service attacks, the directory service allows setting a limit on the size of custom $DirQL_{SE}$ queries.

Compiled queries are loaded by separate classloaders, which take the bytecode directly from the in-memory representation generated by BCEL. The loaded class is instantiated and cast to the `Ranking` interface that is loaded by the system classloader. The directory implementation (which is loaded by the system classloader as well) accesses the user-defined functions only through the `Ranking` interface.

As service composition clients may use the same selection and ranking function for multiple queries, our directory keeps them in a cache. This cache maps a hashcode of the $DirQL_{SE}$ expression to a structure containing the $DirQL_{SE}$ expression as well as the loaded class. In case of a cache hit the user-defined code is compared with the cache entry, and if it matches, the function in the cache is reused, avoiding compilation and linking. This approach mitigates the overhead of query compilation. Because of the way the code is generated (e.g., no use of synchronization, no use of class or instance fields, etc.) multiple invocations of the same ranking function cannot influence each other. The cache employs a least-recently-used replacement strategy. If a function is removed from the cache, it becomes eligible for garbage collection as soon as it is not in use by any query.

Before a compiled $DirQL_{SE}$ function is used, the directory checks that it will not cause any division by zero. This is done by computing every divisor in the $DirQL_{SE}$ expression. Note that this check is not performed during the compilation, as thanks to the caching the compiled code

[5]Although the overhead of generating bytecode using BCEL is negligible in our case (typical $DirQL_{SE}$ expressions are rather small), the performance of our code generator could be further improved by using a recent bytecode manipulation framework, such as ASM (available at http://asm.objectweb.org/).

may be reused in different queries (`size(qin)` and `size(qout)` may vary in different queries). The compiler only verifies that the divisor will be constant throughout a query.

7. Matchmaking and Service Composition

Research on matching of software components (Zaremski, & Wing, 1997) has considered several possible match types based on the implication relations between preconditions and postconditions of a library component s and a query q. E.g., the *plugin* match requires that the preconditions of q imply all preconditions of s and that the postconditions of s imply all postconditions of q. The LARKS matchmaker (Sycara, Lu, Klusch, & Widoff, 1999) also supports the plugin match, but instead of implication a better tractable operation, the θ subsumption over sets of constraints, is used. Recent work on matchmaking (Paolucci, Kawamura, Payne, & Sycara, 2002; Li, & Horrocks, 2003; Constantinescu, & Faltings, 2003) has extended these approaches by using languages based on description logic (Baader, & Sattler, 2001), such as OWL-S, to describe services and queries. The plugin match can be specified in $DirQL_{SE}$, too, as illustrated in Table 7. However, we use less restrictive matching conditions for service composition, as discussed below and in Section 8.

```
select (and (<= (minus sin qin S_CONTAINS_Q) 0)
            (<= (minus qout sout Q_CONTAINS_S) 0))
```

Table 7. Plugin match expressed in $DirQL_{SE}$.

Our service composition algorithms take as input a query, consisting of a set of provided input parameters (corresponding to `qin` in a directory query) and a set of required output parameters (similar to `qout` in a directory query). The composition algorithms interact with the directory using $DirQL_{SE}$ expressions. If a given service composition problem can be solved, the algorithms returns a workflow that describes how to compose services in order to compute all the outputs required by the query.

Algorithms based on forward chaining iteratively apply a possible service s to a set of input parameters provided by the query q. In order to apply a service s to the inputs provided by a query q using forward chaining, for each input required by s, there has to be a compatible parameter in the inputs provided by the query q. Compatibility has to be achieved for the parameter names (that have to be identical) and for their types, where the range provided by the query q has to be more specific than the one accepted by the service s (`S_CONTAINS_Q`). We call this kind of matching of the provided inputs of q with the required inputs of s *forward chaining with complete type matches*. If applying s does not solve the problem (i.e., still not all the outputs required by the query q are available), a new query q' can be computed from q and s, and the whole process is iterated.

Frequently, forward chaining with complete type matches cannot be applied, because the type of a required service input may not cover the full range of possible query inputs. However, there

may be multiple services that together cover the full range. Thus, we have developed another service composition algorithm based on forward chaining, which we call *forward chaining with partial type matches* (Constantinescu, Faltings, & Binder, 2004d). In this algorithm we do not require the range of a service input parameter to cover the full range of an available query input parameter. For a service to be considered, it is sufficient that for each required service input there is a parameter with the same name and overlapping type provided by the query (OVERLAP).

In the case of backward chaining, we start from the set of parameters required by the query q and in each step of the process we choose a service s that will provide at least one of the required parameters. Applying s might result in new parameters being required, resulting in a new query q'. The process is iterated until a solution is found or the composition fails.

A detailed presentation of our service composition algorithms can be found in (Constantinescu, Faltings, & Binder, 2004d).

8. Example Queries for Service Composition

In this section we discuss two simple selection and ranking heuristics and illustrate their compilation: The first one is suited for service composition algorithms using forward chaining, the second one for algorithms based on backward chaining.

```
select (and (<= (minus sin qin S_CONTAINS_Q) 0)
            (> (minus sout qin Q_CONTAINS_S) 0))
order by asc (minus qout sout Q_CONTAINS_S)
```

(a) User-defined selection and ranking function.

```
select (> (minus sout qin Q_CONTAINS_S) 0)
order by asc (minus qout sout OVERLAP)
```

(b) Generated code for inner nodes.

```
final class RankingForwardComplete implements Ranking {

   double rankLeaf( ParamSet qin, ParamSet qout,
                    ParamSet sin, ParamSet sout ) {
      if (sin.minus(qin, ParamSet.S_CONTAINS_Q) > 0 ||
          sout.minus(qin, ParamSet.Q_CONTAINS_S) <= 0)
         return -1.0d;
      return (double)qout.minus(sout, ParamSet.Q_CONTAINS_S);
   }

   double rankInner( ParamSet qin, ParamSet qout,
                     ParamSet sin, ParamSet sout ) {
      if (sout.minus(qin, ParamSet.Q_CONTAINS_S) <= 0)
         return -1.0d;
      return (double)qout.minus(sout, ParamSet.OVERLAP);
   }
}
```

(c) Compiled selection and ranking function.

Table 8. Forward chaining (complete matches).

For forward chaining with complete type matches (see Table 8 (a)), we want that all inputs required by the service are provided by the query (and the service has to be able to handle the parameter types of the provided inputs, i.e., the types in the query have to be more specific than in the service). Moreover, we require that the service provides new outputs which are not already available as query inputs. The results are sorted in ascending order according to the remaining outputs that are required by the query, but not provided by the service (services that provide more of the required outputs come first). In order to support partial type matches, only `S_CONTAINS_Q` has to be replaced with `OVERLAP` in the first line of the selection expression in Table 8 (a).

For backward chaining (see Table 9 (a)), we expect that the service provides at least one output that is required by the query. The results are sorted in ascending order according to the number of missing parameters after application of the service, i.e., the missing inputs of the service and the missing outputs as required by the query.

```
select (> (intersection qout sout Q_CONTAINS_S) 0)
order by asc (+ (minus sin qin S_CONTAINS_Q)
                (minus qout sout Q_CONTAINS_S) )
```

(a) User-defined selection and ranking function.

```
select (> (intersection qout sout OVERLAP) 0)
order by asc (minus qout sout OVERLAP)
```

(b) Generated code for inner nodes.

```
final class RankingBackward implements Ranking {

   double rankLeaf( ParamSet qin, ParamSet qout,
                    ParamSet sin, ParamSet sout ) {
      if (qout.intersection(sout, ParamSet.Q_CONTAINS_S) <= 0)
         return -1.0d;
      return (double)(sin.minus(qin, ParamSet.S_CONTAINS_Q) +
                      qout.minus(sout, ParamSet.Q_CONTAINS_S));
   }

   double rankInner( ParamSet qin, ParamSet qout,
                     ParamSet sin, ParamSet sout ) {
      if (qout.intersection(sout, ParamSet.OVERLAP) <= 0)
         return -1.0d;
      return (double)qout.minus(sout, ParamSet.OVERLAP);
   }
}
```

(c) Compiled selection and ranking function.

Table 9. Backward chaining.

The code for inner nodes is generated according to the transformation scheme presented in the previous section, as illustrated in Table 8 (b) and Table 9 (b). Note that after applying the transformation rules, the resulting expressions have been simplified according to simple algebraic rules, such as '`(<= 0 0)` = *true*', '`(and` *true* *X* `)` = *X*', '`(+ 0` *X* `)` = *X*', etc.

Table 8 (c) and Table 9 (c) illustrate the compilation of these exemplary selection and ranking functions according to the scheme presented in Table 6. For the sake of better readability, we show the compiled methods as Java code, whereas our implementation works at the JVM bytecode level and does not involve the generation of Java code.

9. Evaluation and Future Work

We evaluated the effectiveness of the best-first search with our service composition testbed (Constantinescu, Faltings, & Binder, 2004c). The directory was filled with random service descriptions and we executed service composition algorithms on random composition problems (queries). We used two service composition algorithms based on forward chaining, one that handles only complete type matches, and a second one that can compose partially matching services, too.

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. This corresponds to a previous version of our directory (Constantinescu, Binder, & Faltings, 2004b). In the second configuration we evaluated the new directory that performs a *best-first search* applying the transformed selection 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 for each iteration, we ran the composition algorithms for the same set of random composition problems.

Figure 3 shows the benefits of the best-first search. The Y-axis represents the average percentage of visited directory nodes for the best-first search, relative to the full search. Independent of the directory size and for both service composition algorithms, the best-first search visits only 20–40% of the nodes that are processed by the full search.

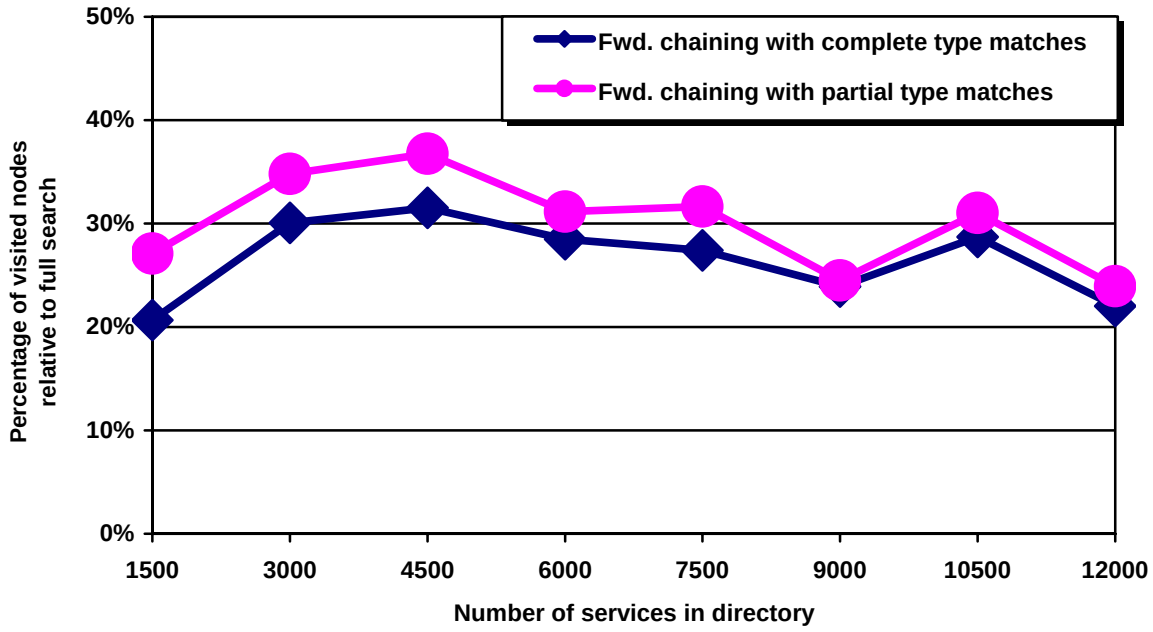

Figure 3. Average percentage of visited directory nodes for best-first search, relative to full search (full search = 100%).

Concerning future work, we are developing service composition algorithms using backward chaining, which tend to be more goal-oriented than approaches based on forward chaining. This can be seen in Table 9 (a), where the selection expression for backward chaining specifies that a matching service must provide at least one output required by the query. Table 9 (b) shows that this criterion is preserved in the transformed selection expression for inner nodes, whereas in the case of forward chaining (see Table 8), the selection expression for inner nodes loses one important condition (which cannot be decided in inner nodes). Hence, we expect the best-first search to yield an even stronger pruning for composition algorithms using backward chaining.

10. Conclusion

Efficient service composition in an open environment populated by a large number of services requires a highly optimized interaction between large-scale directories and service composition engines. Our directory service addresses this need with special features for service composition: Indexing techniques allowing the efficient retrieval of (partially) matching services, incremental data retrieval, as well as *user-defined selection and ranking functions* to support application-specific search heuristics within the directory. The selection and ranking functions are written in a simple, *declarative language*. For efficient search, they are *automatically transformed* in order to enable a *best-first search* that generates the result set lazily, reducing response time and workload within the directory. Moreover, the directory *compiles* user-defined selection and ranking functions to speed up the directory search. Performance measurements confirm that the best-first search effectively prunes the search space.

References

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

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

Benatallah, B., Hacid, M., Rey, C., & Toumani, F. (2003). Semantic reasoning for web services discovery. In *WWW 2003 Workshop on E-Services and the Semantic Web*, Budapest, Hungary.

Benatallah, B., Sheng, Q. Z., & Dumas, M. (2003). The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48.

Binder, W., Constantinescu, I., & Faltings, B. (2004). A directory for web service integration supporting custom query pruning and ranking. In *European Conference on Web Services (ECOWS 2004)*, pages 87–101, Erfurt, Germany.

BPEL4WS. Business process execution language for web services version 1.1. http://www.ibm.com/developerworks/library/ws-bpel/

Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., & Shan, M.-C. (2000). Adaptive and dynamic service composition in eFlow. Technical Report HPL-2000-39, Hewlett Packard Laboratories.

Constantinescu, I., Binder, W., & Faltings, B. (2004a). Directory services for incremental service integration. In *First European Semantic Web Symposium (ESWS-2004)*, pages 254–268, Heraklion, Greece.

Constantinescu, I., Binder, W., & Faltings, B. (2004b). An extensible directory enabling efficient semantic web service integration. In *rd International Semantic Web Conference (ISWC 2004)*, Hiroshima, Japan.

Constantinescu, I., Binder, W., & Faltings, B. (2005). Flexible and efficient matchmaking and ranking in service directories. In *IEEE International Conference on Web Services (ICWS-2005)*, Florida.

Constantinescu, I., & Faltings, B. (2003). Efficient matchmaking and directory services. In *The 2003 IEEE/WIC International Conference on Web Intelligence*, pages 75–81.

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

Constantinescu, I., Faltings, B., & Binder, W. (2004d). Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS-2004)*, pages 506–513, San Diego, CA, USA.

Dahm, M. (1999). Byte code engineering. In *Java-Information-Tage 1999 (JIT'99)*. http://jakarta.apache.org/bcel/

Fuggetta, A., Picco, G. P., & Vigna, G. (1998). Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361.

Giunchiglia, F., & Traverso, P. (1999). Planning as model checking. In *European Conference on Planning*, pages 1–20.

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

Kawamura, T., Blasio, J.-A. D., Hasegawa, T., Paolucci, M., & Sycara, K. P. (2003). Preliminary report of public experiment of semantic service matchmaker with UDDI business registry. In *ICSOC*, pages 208–224.

Lassila, O., & Dixit, S. (2004). Interleaving discovery and composition for simple workflows. In *Semantic Web Services, 2004 AAAI Spring Symposium Series.*

Li, L., & 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., Son, T., & Zeng, H. (2001). Mobilizing the semantic web with DAML-enabled web services. In *Proc. Second International Workshop on the Semantic Web (SemWeb-2001)*, Hongkong.

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

OWL-S. DAML Services. http://www.daml.org/services/owl-s/

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

Ponnekanti, S. R., & Fox, A. (2002). Sword: A developer toolkit for web service composition. In *th World Wide Web Conference (Web Engineering Track)*.

Sun Microsystems, Inc. Java HotSpot Technology. http://java.sun.com/products/hotspot/

Sycara, K., Lu, J., Klusch, M., & Widoff, S. (1999). Matchmaking among heterogeneous agents on the internet. In *Proceedings of the 1999 AAAI Spring Symposium on Intelligent Agents in Cyberspace*, Stanford University, USA.

Traverso, P., & Pistore, M. (2004). Automated composition of semantic web services into executable processes. In *International Semantic Web Conference*, volume 3298 of *Lecture Notes in Computer Science*, pages 380–394. Springer.

UDDI. Universal Description, Discovery and Integration Web Site. http://www.uddi.org/

Verma, K., Sivashanmugam, K., Sheth, A., Patil, A., Oundhakar, S., & Miller, J. (2004). METEOR-S WSDI: A Scalable Infrastructure of Registries for Semantic Publication and Discovery of Web Services. *Journal of Information Technology and Management*.

W3C. OWL Web Ontology Language 1.0 Reference. http://www.w3.org/tr/owl-ref/

W3C. Web services description language (WSDL) version 1.2. http://www.w3.org/tr/wsdl12

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

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

ABOUT THE AUTHORS

Walter Binder is assistant professor at the Faculty of Informatics, University of Lugano. Before joining the University of Lugano, he was a senior researcher at the Artificial Intelligence Laboratory, Ecole Polytechnique Fédérale de Lausanne (EPFL). He holds a Ph.D. in computer science from the Vienna University of Technology. His main research interests are in the area of service-oriented computing, virtual execution environments, and middleware.

Ion Constantinescu holds a Ph.D. in computer science from the Ecole Polytechnique Fédérale de Lausanne (EPFL). His research interests are in the area of service-oriented computing.

Boi Faltings is professor of computer science at the Ecole Polytechnique Fédérale de Lausanne (EPFL), where he heads the Artificial Intelligence Laboratory. He holds a doctorate in Artificial Intelligence from the University of Illinois and has been active in various areas of AI, most recently in constraint programming and multi-agent systems. He is interested in web services as a means for implementing distributed multi-agent systems.