Directory Support for Large-Scale, Automated Service Composition

Walter Binder, Ion Constantinescu, and Boi Faltings

Ecole Polytechnique Fédérale de Lausanne (EPFL),
Artificial Intelligence Laboratory,
CH-1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

Abstract. In an open environment populated by large numbers of services, automated service composition is a major challenge. In such a setting the efficient interaction of directory-based service discovery with different service composition algorithms is crucial. In this paper we present a directory with dedicated features for 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 in order to enable a best-first search for matching directory entries, efficiently pruning the search space.[1]

Keywords: Service discovery and composition, Service directories, Query language and query processing.

1 Introduction

There is a good body of work which addresses the service composition problem by applying planning techniques based either on theorem proving (e.g., Golog [6]) or on hierarchical task planning (e.g., SHOP-2 [7]). 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 have to be adapted to a situation where planning operators are not known a priori, but have to be retrieved through queries to these directories.

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 computing and introducing *switches* in the composi-

[1] The work presented in this paper 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).

tion plan. Experimental results show that using partial matches significantly decreases the failure rate compared with a composition algorithm that supports only complete matches [4].

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. 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 [2].

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 paper, 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 [2,1], the novel, original contributions of this paper are the declarative directory query language and the transformation mechanism to make better use of the internal directory structure. These techniques, which have not been applied in the context of service directories before, provide a flexible and efficient mechanism for query processing.

This paper is structured as follows: Section 2 gives an overview of our service description formalism and of the internal index structure of our directory. In Section 3 we present a simple, functional query language which allows to express application-specific selection and ranking heuristics. Section 4 explains the processing of directory queries and introduces query transformations that enable a best-first search with early pruning. In Section 5 we discuss how user-defined queries are compiled and integrated into the directory. Section 6 discusses a sample query and shows its transformation. Finally, Section 7 concludes this paper.

2 Service Descriptions and Directory Index

Service descriptions are a key element for service discovery and service composition and should enable automated interactions between applications. In this paper we partially build on existing formalisms, such as WSDL (http://www.w3.org/TR/wsdl) and OWL-S (http://www.daml.org/services/owl-s/), by considering a simple table-based formalism where each service is described by a set of tuples mapping service parameters (unique names of inputs or outputs) to parameter types (the spaces of possible values for a given 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 expressed either as sets of intervals of basic data types (e.g., date/time, integers, floating-points) or as classes of individuals. Class parameter types can be defined in a descriptive language such as OWL (http://www.w3.org/2004/OWL/). From the descriptions we derive a directed graph (DG) of simple is-a relations either directly (for basic data types) or by using a description logic classifier (for concepts). For efficiency reasons, we represent the DG numerically. We assume that each class is represented as a set of intervals. We encode each parent-child relation by sub-dividing each of the intervals of the parent. In the case of multiple parents, the child class is represented by the union of the sub-intervals resulting from the encoding of each of the parent-child relations. Since for a given domain we can have several parameters represented by intervals, the space of all possible parameter values can be represented as a rectangular hyperspace with a dimension for each parameter. For details, see [3].

The need for efficient discovery and matchmaking leads to a need for search structures and indexes for directories. We consider numerically encoded 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 [5]. 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.

Each leaf node in the GiST of our directory holds references to all service descriptions with a certain input/output behaviour. 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, the union of all inputs/outputs found in the subtree is stored. More precisely, 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. That is, 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.

3 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 [2]. 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.

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.[2] 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 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 4).
- Efficient compilation: Due to the simplicity of the language, $DirQL_{SE}$ expressions can be efficiently compiled to increase performance (see Section 5).

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 (encoded as a set of numeric intervals).

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

[2] In order to simplify the presentation, in this paper 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.

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

```
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' ;
```

second argument is bigger than the first one. The $DirQL_{SE}$ programmer may use the '`if`' conditional to ensure that the first argument of '`-`' is bigger or equal than the second one. For division, the second operand (divisor) has to evaluate to a constant for a given query. That is, it is a numeric expression with only numeric constants, as well as `size(qin)` and `size(qout)` at the leaves. 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).

4 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 a very efficient generation of the selection and ranking functions for inner nodes (the execution time of the transformation algorithm is linear with the size of the query).

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 priority queue or the number of retrieved results exceed a certain threshold defined by the directory service provider.

Table 2 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

Table 2. Transformation operators $\uparrow$, $\downarrow$, $\oplus$, and $\ominus$ for the generation of inner node code

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

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` $\uparrow E$'.

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. While a formal proof of the correctness of the transformation rules in Table 2 had to be omitted due to space limitations, we exemplarily explain 2 rules in an informal 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 2, $\oplus t$ 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.

Table 3. Transformation operator $\updownarrow$ for the generation of code in inner nodes of the GiST

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

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 intersection, 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 2, $\ominus t$ 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`)`'.

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 '$\uparrow(/\ a\ c) \longrightarrow (/\ \uparrow a\ \downarrow c)$'. 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 ($\downarrow c = 0$). 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 3). 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 3 a and b are numeric expressions, while x and y are boolean expressions. Again, due to space limitations, a formal proof of these rules cannot be included in this paper.

The alert reader may have noticed that the operators '`not`' and '`=`' have been omitted in Table 3. 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. 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`))`.

5 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. The compiler uses the Bytecode Engineering Library BCEL (`http://jakarta.apache.org/bcel/`) to manipulate JVM bytecode.

Compiling and integrating user-defined code into the directory leverages state-of-the-art optimizations in recent JVM implementations. Many modern JVMs first interpret bytecode to gather 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. Due to space limitations, details concerning the compilation of $DirQL_{SE}$ expressions had to be omitted.

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.

6 Example Query for Service Composition

In this section we show the transformation of a simple selection and ranking function for service composition based on forward chaining [4].

For forward chaining with complete type matches (see Table 4 (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 [4], only `S_CONTAINS_Q` has to be replaced with `OVERLAP` in the first line of the selection expression in Table 4 (a).

The code for inner nodes is generated according to the transformation scheme presented in Section 4, as illustrated in Table 4 (b). Note that after applying the transforma-

Table 4. Selection and ranking function for service composition using forward 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.

tion rules, the resulting expressions have been simplified according to simple algebraic rules, such as '`(<= 0 0) =` *true*', '`(and` *true X*`) =` *X*', etc.

7 Conclusion

In this paper we presented a service directory with special support 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 that enable the dynamic installation of application-specific heuristics within the directory. In order to efficiently support different service composition algorithms, it is important not to hard-code such heuristics in the directory, but to enable the dynamic installation of specific pruning and ranking heuristics. The selection and ranking functions are written in a simple, declarative language. Thanks to the support of application-specific heuristics, the most promising results from a directory query are returned first, which helps to reduce the number of transferred results and to save network bandwidth. Moreover, the result set is generated lazily, reducing response time and the workload in the directory. For efficient execution, the directory transforms and compiles user-defined selection and ranking functions.

References

1. W. Binder, I. Constantinescu, and B. Faltings. 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, Sept. 2004.
2. I. Constantinescu, W. Binder, and B. Faltings. Directory services for incremental service integration. In *First European Semantic Web Symposium (ESWS-2004)*, pages 254–268, Heraklion, Greece, May 2004.
3. I. Constantinescu and B. Faltings. Efficient matchmaking and directory services. In *The 2003 IEEE/WIC International Conference on Web Intelligence*, pages 75–81, 2003.
4. I. Constantinescu, B. Faltings, and W. Binder. Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS-2004)*, pages 506–513, San Diego, CA, USA, July 2004.
5. J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proc. 21st Int. Conf. Very Large Data Bases, VLDB*, pages 562–573. Morgan Kaufmann, 11–15 1995.
6. S. A. McIlraith and T. C. Son. Adapting Golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams, editors, *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 482–496, San Francisco, CA, Apr. 22–25 2002. Morgan Kaufmann Publishers.
7. D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S web services composition using SHOP2. In *Proceedings of 2nd International Semantic Web Conference (ISWC-2003)*, 2003.