A Directory for Web Service Integration Supporting Custom Query Pruning and Ranking

Walter Binder, Ion Constantinescu, and Boi Faltings

Artificial Intelligence Laboratory
Swiss Federal Institute of Technology Lausanne (EPFL)
CH–1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

Abstract. In an open environment populated by large numbers of heterogeneous information services, integration is a major challenge. In such a setting, the efficient coupling between directory-based service discovery and service composition engines is crucial. In this paper we present a directory service that offers specific functionality in order to enable efficient web service integration. Results matching with a directory query are retrieved incrementally on demand, whenever the service composition engine needs new results. In order to optimize the interaction of the directory with different service composition algorithms, the directory supports custom pruning and ranking functions that are dynamically installed with the aid of mobile code. The pruning and ranking functions are written in Java, but the directory service imposes severe restrictions on the programming model in order to protect itself against malicious or erroneous code. With the aid of user-defined pruning and ranking functions, application-specific ordering heuristics can be directly installed into the directory. Due to its extensibility, the directory can be tailored to the needs of various service integration algorithms. This is crucial, as service composition still needs a lot of research and experimentation in order to develop industrial-strength algorithms. Experiments on randomly generated problems show that special pruning and ranking functions significantly reduce the number of query results that have to be transmitted to the client by up to 5 times.[1]

Keywords: Incremental service integration, query pruning, service directories, service discovery, service ranking

1 Introduction

Service composition[2] is an exciting area which has received a significant amount of interest in the last period. Initial approaches to web service composition [25] used a simple forward chaining technique which can result in the discovery of large numbers

[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 Science Foundation as part of the project MAGIC (FNRS-68155), as well as by the Swiss National Funding Agency OFES as part of the European projects KnowledgeWeb (FP6-507482) and DIP (FP6-507483).

[2] In this paper we use the terms 'service composition' and 'service integration' interchangeably.

L.-J. Zhang and M. Jeckle (Eds.): ECOWS 2004, LNCS 3250, pp. 87–101, 2004.

of services. There is a good body of work which tries to address the service composition problem by applying planning techniques based either on theorem proving (e.g., Golog [19,20], SWORD [22]) or on hierarchical task planning (e.g., SHOP-2 [31]). The advantage of this kind of approach is that complex constructs like loops (Golog) or processes (SHOP-2) can be handled. All these approaches assume that the relevant service descriptions are initially loaded into the reasoning engine and that no discovery is performed during composition.

Still the current and future state of affairs regarding web services will be quite different since due to the large number of services and to the loose coupling between service providers and consumers we expect that services will be indexed in directories. Consequently, planning algorithms will have to be adapted to a situation where operators are not known a priori, but have to be retrieved through queries to these directories. Recently, Lassila and Dixit [16] have addressed the problem of interleaving discovery and integration in more detail, but they have considered only simple workflows where services have one input and one output.

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 [10]. Our composition algorithm allows for *partially matching* types and handles them by computing and introducing *switches* in the integration plan. Experimental results carried out in various domains show that using partial matches decreases the failure rate by up to *7 times* compared with an integration algorithm that supports only complete matches [10].

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 [8]. 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 [7]. Sessions are well isolated from each other, also concurrent modifications of the directory (i.e., new service registrations, updates, and removal of services from the directory) do not affect the sequence of results after a query has been issued. We have implemented a simple but very efficient concurrency control scheme which may delay the visibility of directory updates but does not require any synchronization activities within sessions. Hence, our concurrency control mechanism has no negative impacts on the scalability with respect to the number of concurrent sessions.

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 transfer first the better matches to the client. If the ranking 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 ranking heuristics depend on the concrete composition algorithm. For each service composition algorithm (e.g., forward chaining, backward chaining, etc.), a different ranking 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 integration algorithms, our directory supports *user-defined pruning and ranking functions*.

Custom pruning and ranking functions allow the execution of user-defined application-specific heuristics directly within the directory, close to the data, in order to transfer the best results for a query first. They dramatically increase the flexibility of our directory, as the client is able to tailor the processing of directory queries according to its needs. The pruning and ranking functions are written in Java and dynamically installed during service composition sessions by means of *mobile code*. Because the support of mobile code increases the vulnerability of systems and custom ranking functions may be abused for various kinds of attacks, such as denial-of-service attacks consuming a vast amount of memory and processing resources within the directory, our directory imposes severe restrictions on the code of these functions.

As the main contributions of this paper, we show how our directory supports user-defined pruning and ranking functions. We present the restricted programming model for pruning and ranking functions and discuss how they are used within directory queries. Moreover, we explain how our directory protects itself against malicious code. Performance evaluations point up the necessity to support heuristic pruning and ranking functions that are tailored to specific service composition algorithms.

This paper is structured as follows: In Section 2 we explain how service descriptions can be numerically encoded as sets of intervals and we give an overview of the index structure for multidimensional data on which our directory implementation is based. In Section 3 we show how the directory can be dynamically extended by user-defined pruning and ranking functions. We discuss some implementation details, in particular showing how the directory protects itself against malicious code. In Section 4 we present some experimental results that illustrate the need for dynamically installing application-specific, heuristic pruning and ranking functions. Section 5 presents ongoing research and our plans to improve the directory service in the near future. Finally, Section 6 concludes this paper.

2 Directories for Service Descriptions

Since we assume a large-scale open environment with a high number of available services, the integration process has to be able to discover relevant services incrementally through queries to the service directory. Interleaving the integration process with service discovery in a large-scale directory is a novelty of our approach. In this section we give an overview of the basic features of our directory and the used indexing structures.

2.1 Existing Directories

Currently, UDDI (Universal Description, Discovery, and Integration) [26] is the state-of-the-art for directories of web services. UDDI is an industrial effort to create an open specification for directories of service descriptions. It builds on existing technology standardized by the World Wide Web Consortium[3] like the eXtensible Markup Language

[3] `http://www.w3c.org/`

(XML), the Simple Object Access Protocol (SOAP), and the Web Services Description Language (WSDL). The UDDI standard is clear in terms of data models and query API, but suffers from the fact that it considers service descriptions to be completely opaque.

A more complex method for discovering relevant services from a directory of advertisements is matchmaking. In this case the directory query (requested capabilities) is formulated in the form of a service description template that presents all the features of interest. This template is then compared with all the entries in the directory and the results that have features compatible with the features of interest are returned. A good amount of work exists in the area of matchmaking, including LARKS [24] and the newer efforts geared towards OWL-S [21]. Other approaches include the Ariadne mediator [15].

2.2 Match Types

We consider four match relations between a query Q and a service S:

Exact: S is an exact match of Q.
PlugIn: S is a plug-in match for Q, if S could be always used instead of Q.
Subsumes: Q contains S. In this case S could be used under the condition that Q satisfies some additional runtime constraints such that it is specific enough for S.
Overlap: Q and S have a given intersection. In this case, runtime constraints both over Q and S have to be taken into account.

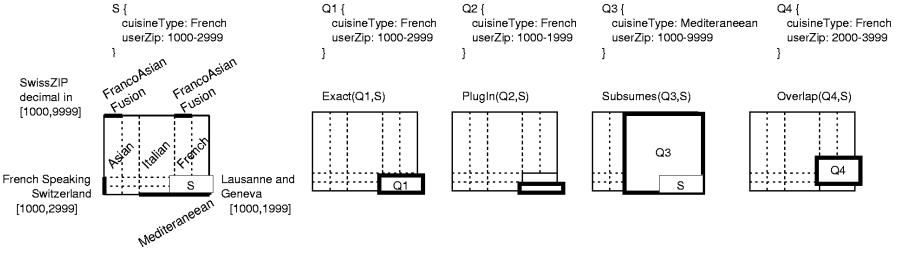

Fig. 1. Match types of inputs of query Q and service S by 'precision': *Exact*, *PlugIn*, *Subsumes*, *Overlap*.

In the example in Fig. 1 we show how the match relation is determined between the inputs available from the queries $Q1$, $Q2$, $Q3$, $Q4$ and the inputs required by the service S. We can order the types of match by 'precision' as follows: *Exact*, *PlugIn*, *Subsumes*, *Overlap*. We consider *Subsumes* and *Overlap* as 'partial' matches. The first three relations have been previously identified by Paolucci in [21] and the fourth, *Overlap*, was identified by Li [17] and Constantinescu [8].

Determining one match relation between a query description and a service description requires that subsequent relations are determined between all the inputs of the query Q and service S and between the outputs of the service S and query Q (note the reversed order of query and services in the match for outputs). Our approach is more complex

than the one of Paolluci in that we take also into account the relations between the properties that introduce different inputs or outputs (equivalent to parameter names). This is important for disambiguating services with equivalent signatures (e.g., we can disambiguate two services that have two string outputs by knowing the names of the respective parameters).

2.3 Numerically Encoding Services and Queries

Service descriptions are a key element for service discovery and service composition and should enable automated interactions between applications. Currently, different overlapping formalisms are proposed (e.g., [29], [26], [11], [12]) and any single choice could be quite controversial due to the trade-off between expressiveness and tractability specific to any of the aforementioned formalisms.

In this paper we partially build on existing developments, such as [29], [1], and [11], 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). 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 like XML Schema [30] or in the Ontology Web Language [28]. From the descriptions we can derive a directed graph (DG) of simple 'is-a' relations either directly or by using a description logic classifier.

For efficiency reasons, we represent the DG numerically. We assume that each class will be represented as a set of intervals. We encode each parent-child relation by subdividing 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. Details concerning the numerical encoding of services can be found in [8].

2.4 Multidimensional Access Methods – GiST

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. This approach leads to local response times in the order of milliseconds for directories containing tens of thousands (10^4) of service descriptions.

The indexing technique that we use is based on the Generalized Search Tree (GiST), proposed as a unifying framework by Hellerstein [14] (see Fig. 2). 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 internal node holds a key in the form of a predicate P and a number of pointers to other nodes (depending on system and hardware constraints, e.g., filesystem page size). To search for records that satisfy a query predicate Q, the paths of the tree that have keys P satisfying Q are followed.

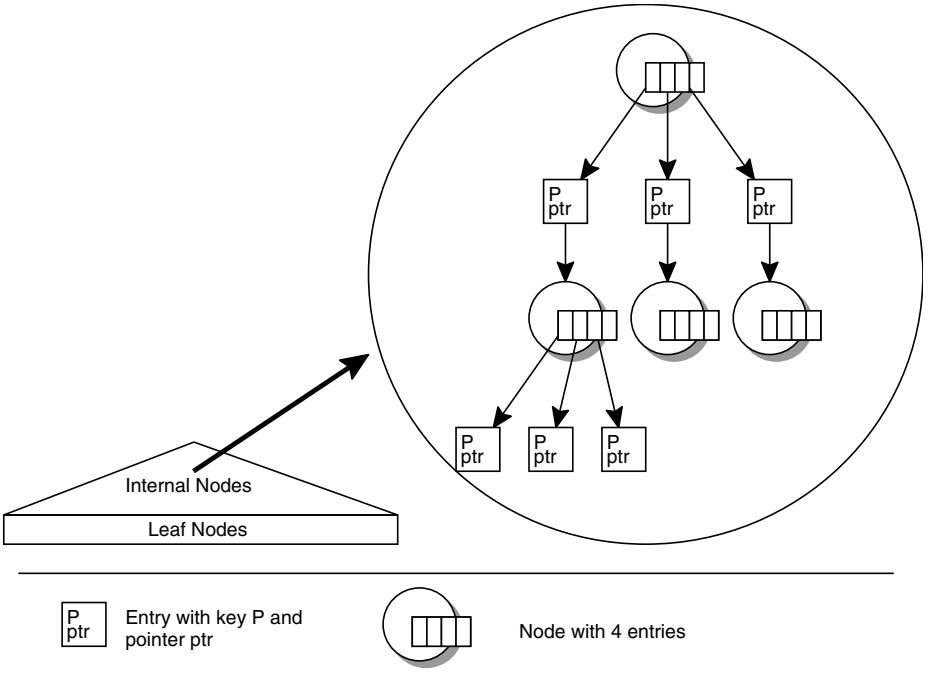

Fig. 2. Generalised Search Tree (GiST).

3 Custom Pruning and Ranking Functions

As directory queries may retrieve large numbers of matching entries (especially when partial matches are taken into consideration), our directory support sessions in order to incrementally access the results of a query [7]. By default, the order in which matching service descriptions are returned depends on the actual structure of the directory index (the GiST structure discussed before). However, depending on the service integration algorithm, ordering the results of a query according to certain heuristics may significantly improve the performance of service composition. In order to avoid the transfer of a large number of service descriptions, the pruning, ranking, and sorting according to application-dependent heuristics should occur directly within the directory. As for each service integration algorithm a different pruning and ranking heuristic may be better suited, our directory allows its clients to define custom pruning 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 [13]. Within a service integration session multiple pruning and ranking functions may be defined. Each query in a session may be associated with a different one.

3.1 API for Pruning and Ranking Functions

The pruning and ranking functions receive as arguments information concerning the matching of a service description in the directory with the current query. They return a

value which represents the quality of the match. The bigger the return value, the better the service description matches the requirements of the query. The sequence of results as returned by the directory is sorted in descending order of the values calculated by the pruning and ranking functions (a higher value means a better match). Results for which the functions evaluate to zero come at the end, a negative value indicates that the match is too poor to be returned, i.e., the result is discarded and not passed to the client (pruning).

As arguments the client-defined pruning and ranking functions take four `ParamSet` objects corresponding to the input and output parameter sets of the query, and respectively of the service. The `ParamSet` object provides methods like size, membership, union, intersection, and difference (see Fig. 3). The size and membership methods require only the current `ParamSet` object, while the union, intersection, and difference methods use two `ParamSet` objects – the current object and a second `ParamSet` object passed as argument.

It is important to note that some of the above methods address two different issues at the same time:

1. Basic *set operations*, where a set member is defined by a parameter name and its type; for deciding the equality of parameters with same name and different types a user-specified expression is used.
2. *Computation of new types* for some parameters in the resulting sets; when a parameter is common to the two argument sets its type in the resulting set is computed with a user-specified expression.

The explicit behavior of the `ParamSet` methods is the following:

size: Returns the number of parameters in the current set.

containsParam: Returns true if the current set contains a parameter with the same name as the method argument (regardless of its type).

union: Returns the union of the parameters in the two sets. For each parameter that is common to the two argument sets the type in the resulting set is computed according to the user-specified expression `newTypeExpr`.

intersection: Returns the parameters that are common to the two sets *AND* for which the respective types conform to the equality test specified by the `eqTestExpr`. The type of these parameters in the resulting set is computed according to the user-specified expression `newTypeExpr`.

minus: Returns the parameters that are present only in the current set and in the case of common parameters only those that *DO NOT* conform to the equality test specified by the `eqTestExpr`. For the latter kind of parameters the type in the resulting set is computed according to the user-specified expression `newTypeExpr`.

Parameters whose `newTypeExpr` would be the empty type (called `NOTHING` in the table below) are removed from the resulting set.

The expressions used in the `eqTestExpr` and `newTypeExpr` parameters have the same format and they are applied to parameters that are common to the two `ParamSet` objects passed to a `union`, `intersection`, or `minus` method. For such kind of parameter we denote its type in the two argument sets as `A` and `B`. The expressions are created

from these two types by using some extra constructors based on the Description Logic language OWL [28] like $\top$, $\bot$, $\neg$, $\sqcap$, $\sqcup$, $\sqsubseteq$, $\equiv$. The expressions are built by specifying a constructor type and which of the argument types `A` or `B` should be negated. For the single type constructors $\top$ and $\bot$ negation cannot be specified and for the constructors A and B the negation is allowed only for the respective type (e.g., for the constructor type A, only $\neg A$ can be set).

Constructor type	$\neg A$?	$\neg B$?	Possible expressions
`THING`	-	-	$\top$
`NOTHING`	-	-	$\bot$
`A`	Y/N	-	$A, \neg A$
`B`	-	Y/N	$B, \neg B$
`UNION`	Y/N	Y/N	$A \sqcup B$, $A \sqcup \neg B$, $\neg A \sqcup B$, $\neg A \sqcup \neg B$
`INTERSECTION`	Y/N	Y/N	$A \sqcap B$, $A \sqcap \neg B$, $\neg A \sqcap B$, $\neg A \sqcap \neg B$
`SUBCLASS`	Y/N	Y/N	$A \sqsupseteq B$, $A \sqsupseteq \neg B$, $\neg A \sqsupseteq B$, $\neg A \sqsupseteq \neg B$
`SUPERCLASS`	Y/N	Y/N	$A \sqsubseteq B$, $A \sqsubseteq \neg B$, $\neg A \sqsubseteq B$, $\neg A \sqsubseteq \neg B$
`SAMECLASS`	Y/N	Y/N	$A \equiv B$, $A \equiv \neg B$, $\neg A \equiv B$, $\neg A \equiv \neg B$

We represent an expression as a bit vector having a value corresponding to its respective constructor type. For encoding the negation of any of the types that are arguments to the constructor, two masks can be applied to the constructor types: `NEG_A` and `NEG_B`. For the actual encoding, see Fig. 3. For example, $A \sqcap \neg B$ will be expressed as `ParamSet.INTERSECTION | ParamSet.NEG_B`.

As an example of API usage, assume we need to select the parameters that are common to the two sets X and Y, which have in X a type that is more specific than the one in Y. In the result set we would like to preserve the type values in X. The following statement can be used for this purpose: `X.intersection(Y, ParamSet.SUPERCLASS, ParamSet.A)`.

The directory supports pruning and ranking functions written in a subset of the Java programming language. The code of the functions is provided as a compiled Java class. The class has to implement the `Ranking` interface shown in Fig. 3.

Processing a user query requires traversing the GiST structure of the directory starting from the root node. While `rankInner()` is invoked for inner nodes of the directory tree, `rankLeaf()` is called on leaf nodes. `rankLeaf()` receives as arguments `sin` and `sout` the exact parameter sets as defined by the service description stored in the directory. Hence, `rankLeaf()` has to return a concrete heuristic value for the given service description. In contrast, `rankInner()` receives as arguments `sin` and `sout` supersets of the input/output parameters found in any leaf node of its subtree. The type of each parameter is a supertype of the parameter found in any leaf node (which has the parameter) in the subtree. `rankInner()` has to return a heuristic value which is bigger or equal than all possible ranking values in the subtree. That is, for an inner node the heuristic function has to return an upper bound of the best ranking value that could be found in the subtree. If the upper bound of the heuristic ranking value in the subtree cannot be determined, `Double.POSITIVE_INFINITY` may be used.

The pruning and ranking functions enable the lazy generation of the result set based on a *best-first search* where the visited nodes of the GiST are maintained in a heap or

```
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 THING=1, NOTHING=2, A=3, B=4, UNION=5, INTERSECTION=6,
                   SUBCLASS=7, SUPERCLASS=8, SAMECLASS=9, NEG_A=16, NEG_B=32;

  int size();
  boolean containsParam( String paramName );
  ParamSet union( ParamSet p, int newTypeExpr );
  ParamSet minus( ParamSet p, int eqTestExpr, int newTypeExpr );
  ParamSet intersection( ParamSet p, int eqTestExpr, int newTypeExpr );
}
```

Fig. 3. The API for ranking functions.

priority queue and the most promising one is expanded. If the most promising node is a leaf node, it can be returned. 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 approaches 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.

3.2 Exemplary Ranking Functions

In the example in Fig. 4 two basic ranking functions are shown, the first one more appropriate for service composition algorithms using forward chaining (considering only complete matches), the second for algorithms using backward chaining. Note the weakening of the pruning conditions for the inner nodes.

3.3 Safe and Efficient Execution of Ranking Functions

Using a subset of Java as programming language for pruning and ranking functions has several advantages: Java is well known to many programmers, there are lots of programming tools for Java, and, above all, it integrates very well with our directory service, which is completely written in Java.

Compiling and integrating user-defined ranking functions into the directory leverages state-of-the-art optimizations in recent JVM implementations. For instance, the HotSpot VM [23] first interprets JVM bytecode [18] 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 pruning and ranking functions are executed as efficiently as algorithms directly built into the directory.

```
public final class RankingForward implements Ranking {
  public double rankLeaf( ParamSet qin, ParamSet qout,
                          ParamSet sin, ParamSet sout ) {
    // discard service if it requires parameters that are not in the query;
    // the provided input has to be more specific than the required one
    if (sin.minus(qin, ParamSet.SUBCLASS, ParamSet.A).size() > 0) return -1.0d;

    // services that provide more required parameters are better;
    // the provided output has to be more specific than the required one
    return (double)
             sout.intersection(qout, ParamSet.SUPERCLASS, ParamSet.A).size();
  }

  public double rankInner( ParamSet qin, ParamSet qout,
                           ParamSet sin, ParamSet sout ) {
    // for forward chaining, pruning inner nodes is not possible,
    // but an upper bound of the overlap of the outputs can be easily computed
    return (double)
             sout.intersection(qout, ParamSet.INTERSECTION, ParamSet.A).size();
  }
}

public final class RankingBackward implements Ranking {
  public double rankLeaf( ParamSet qin, ParamSet qout,
                          ParamSet sin, ParamSet sout ) {
    // discard service if it does not provide any required output
    if (sout.intersection(qout, ParamSet.SUPERCLASS, ParamSet.A).size() == 0)
      return -1.0d;

    // services that reduce most the number of required outputs are better
    ParamSet remaining = qout.minus(sout, ParamSet.SUBCLASS, ParamSet.A);
    ParamSet newRequired = sin.minus(qin, ParamSet.SUBCLASS, ParamSet.A);
    ParamSet required = remaining.union(newRequired, ParamSet.INTERSECTION);
    return 1 / (double)(1+required.size());
  }

  public double rankInner( ParamSet qin, ParamSet qout,
                           ParamSet sin, ParamSet sout ) {
    if (sout.intersection(qout, ParamSet.INTERSECTION, ParamSet.A).size() == 0)
      return -1.0d;

    ParamSet remaining = qout.minus(sout, ParamSet.INTERSECTION, ParamSet.A);
    return 1 / (double)(1+remaining.size());
  }
}
```

Fig. 4. Exemplary pruning and ranking functions.

The class containing the ranking function is analyzed by our special bytecode verifier which ensures that the user-defined ranking function always terminates within a well-defined time span and does not interfere with the directory implementation. Efficient, extended bytecode verification to enforce restrictions on JVM bytecode for the safe execution of untrusted mobile code has been studied in the JavaSeal [27] and in the J-SEAL2 [2,5] mobile object kernels. Our bytecode verifier ensures the following conditions:

- The `Ranking` interface is implemented.
- Only the methods of the `Ranking` interface are provided.

- The control-flow graphs of the `rankLeaf()` and `rankInner()` methods are acyclic. The control-flow graphs are created by an efficient algorithm with execution time linear with the number of JVM instructions in the method.
- No exception handlers (using malformed exception handlers, certain infinite loops can be constructed that are not detected by the standard Java verifier, as shown in [6]). If the ranking function throws an exception (e.g., due to a division by zero), its result is to be considered zero by the directory.
- No JVM subroutines (they may result from the compilation of `finally{}` clauses).
- No explicit object allocation. As there is some implicit object allocation in the set operations in `ParamSet`, the number of set operations and the maximum size of the resulting sets are limited.
- Only the interface methods of `ParamSet` may be invoked, as well as a well-defined set of methods from the standard mathematics package.
- Only the static fields defined in the interface `ParamSet` may be accessed.
- No fields are defined.
- No synchronization instructions.

These restrictions ensure that the execution time of the custom pruning and ranking function is bounded by the size of its code. Hence, an attacker cannot crash the directory by providing, for example, a pruning and ranking function that contains an endless loop. Moreover, these functions cannot allocate memory. Our extended bytecode verification algorithm is highly efficient, its performance is linear with the size of the pruning and ranking methods. As a prevention against denial-of-service attacks, our directory service allows to set a limit for the size of custom functions.

Pruning and ranking functions are loaded by separate classloaders, in order to support multiple versions of classes with the same name (avoiding name clashes between multiple clients) and to enable garbage collection of the class structures. The loaded class is instantiated and casted to the `Ranking` interface that is loaded by the system classloader. The directory implementation (which is loaded by the system classloader) accesses the user-defined functions only through the `Ranking` interface.

As service integration clients may use the same ranking functions in multiple sessions, our directory keeps a cache of ranking functions. This cache maps a hashcode of the function class to a structure containing the function bytecode as well as the loaded class. In case of a cache hit the user-defined function code is compared with the cache entry, and if it matches, the function in the cache is reused, skipping verification and avoiding to reload it with a separate classloader. Due to the restrictions mentioned before, 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 service integration session.

4 Evaluation

We have evaluated our approach by carrying out tests on random service descriptions and service composition problems that were generated as described in [9]. As we consider

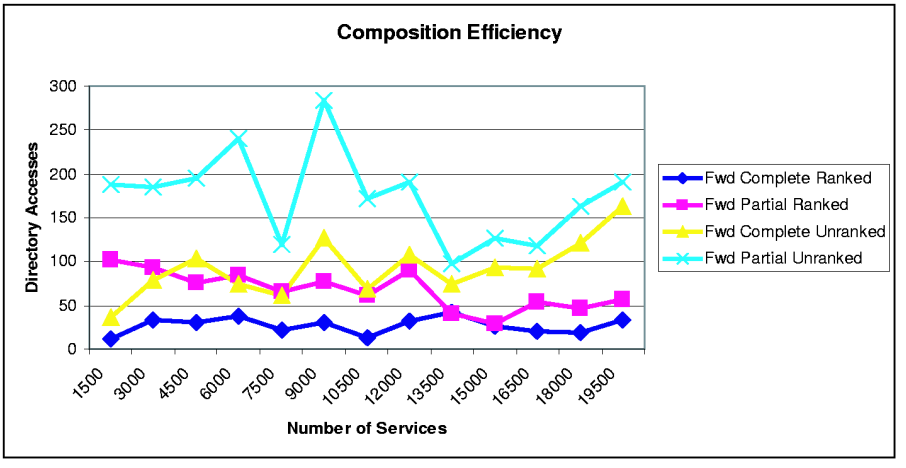

Fig. 5. Impact of ranking functions on composition algorithms.

directory accesses to be a computationally expensive operation we use them as a measure of efficiency.

The problems have been solved using two forward chaining composition algorithms: One that handles only complete type matches and another one that can compose partially matching services, too [10]. When running the algorithms we have used two different directory configuration: The first configuration was using the extensible directory described in this paper which supports custom pruning and ranking functions, in particular using the forward chaining ranking function described in Fig. 4. In the second configuration we used a directory which is not aware of the service composition algorithm (e.g., forward complete, backward, etc.) and cannot be extended by client code. This directory implements a generic ordering heuristic by considering the number of overlapping inputs in the query and in the service, plus the number of overlapping outputs in the query and in the service.

For both directories we have used exactly the same set of service descriptions and at each iteration we have run the algorithms on exactly the same random problems. As it can be seen in Fig. 5, using custom pruning and ranking functions consistently improves the performance of our algorithms. In the case of complete matches the improvement is up to a factor of 5 (for a directory of 10500 services) and in the case of partial matches the improvement is of a factor of 3 (for a directory of 9000 of services).

5 Future Work

As discussed in Section 3, currently the directory enforces severe restrictions on the pruning and ranking functions in order to protect itself against malicious or erroneous code. For instance, the code has to be sequential and it must not explicitly allocate memory.

We are trying to relax these restrictions with the aid of resource management mechanisms, granting a certain amount of memory and CPU cycles for the execution of the pruning and ranking function which will be stopped if they try to exceed their allowed quota. For this purpose we will use J-RAF2[4], The Java Resource Accounting Framework, Second Edition, which offers a fully portable resource management layer for Java [5,3, 4].

J-RAF2 rewrites the bytecode of Java classes before they are loaded and transforms the program in order to expose details concerning its resource consumption during execution. Currently, J-RAF2 addresses memory and CPU control. For memory control, object allocations are intercepted in order to verify that no memory limit is exceeded. For CPU control, the number of executed bytecode instructions are counted and periodically the system checks whether a running thread exceeds its granted CPU quota. J-RAF2 has been successfully tested in standard J2SE, J2ME, and J2EE environments. Due to special implementation techniques, the overhead for resource management is reasonably small, about 20–30%, and only the code rewritten for resource management (i.e., the custom pruning and ranking function) incurs this overhead.

Based on J-RAF2, we will be able to remove certain restrictions on the programming model of pruning and ranking functions. For instance, loops will be allowed in the control flow. Enriching the programming model will also allow the directory to expose a more flexible API to the custom pruning and ranking functions.

In addition to this security-related issue, we are working on a high-level declarative query language, which will ease the specification of query heuristics. We will use on-the-fly compilation techniques to generate Java bytecode for customizing the traversal of the directory tree.

Yet another issue we are currently working on is the distribution of the directory service within a network, in order to increase its availability and scalability and to prevent it from becoming a central point of failure and performance bottleneck.

6 Conclusion

Efficient service integration in an open environment populated by a large number of services requires a highly optimized interaction between large-scale directories and service integration engines. Our directory service addresses this need with special features for supporting service composition: indexing techniques allowing the efficient retrieval of (partially) matching services, service integration sessions offering incremental data retrieval, as well as user-defined pruning and ranking functions that enable the installation of application-specific ranking heuristics within the directory. In order to efficiently support different service composition algorithms, it is important not to hard-code ordering heuristics into the directory, but to enable the dynamic installation of specific pruning and ranking heuristics. Thanks to the custom pruning and ranking functions, 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. As user-defined pruning and ranking functions may be abused to launch denial-of-service attacks

[4] `http://www.jraf2.org/`

against the directory, we impose severe restrictions on the accepted code. Performance measurements underline the need for applying application specific heuristics to order the results that are returned by a directory query.

References

1. D.-S. C. A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. DAML-S: Web service description for the Semantic Web. *Lecture Notes in Computer Science*, 2342, 2002.
2. W. Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, San Diego, CA, USA, Jan. 2001.
3. W. Binder and V. Calderon. Creating a resource-aware JDK. In *ECOOP 2002 Workshop on Resource Management for Safe Languages*, Malaga, Spain, June 2002.
4. W. Binder and J. Hulaas. Self-accounting as principle for portable CPU control in Java. In *5th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays'2004)*, Erfurt, Germany, Sept. 2004.
5. W. Binder, J. Hulaas, A. Villazón, and R. Vidal. Portable resource control in Java: The J-SEAL2 approach. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-2001)*, Tampa Bay, Florida, USA, Oct. 2001.
6. W. Binder and V. Roth. Secure mobile agent systems using Java: Where are we heading? In *Seventeenth ACM Symposium on Applied Computing (SAC-2002)*, Madrid, Spain, Mar. 2002.
7. I. Constantinescu, W. Binder, and B. Faltings. Directory services for incremental service integration. In *First European Semantic Web Symposium (ESWS-2004)*, Heraklion, Greece, May 2004.
8. I. Constantinescu and B. Faltings. Efficient matchmaking and directory services. In *The 2003 IEEE/WIC International Conference on Web Intelligence*, 2003.
9. I. Constantinescu, B. Faltings, and W. Binder. Large scale testbed for type compatible service composition. In *ICAPS 04 workshop on planning and scheduling for web and grid services*, 2004.
10. I. Constantinescu, B. Faltings, and W. Binder. Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS-2004)*, San Diego, CA, USA, July 2004.
11. DAML-S. DAML Services, http://www.daml.org/services.
12. FIPA. Foundation for Intelligent Physical Agents Web Site, http://www.fipa.org/.
13. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.
14. 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.
15. C. A. Knoblock, S. Minton, J. L. Ambite, N. Ashish, I. Muslea, A. Philpot, and S. Tejada. The Ariadne Approach to Web-Based Information Integration. *International Journal of Cooperative Information Systems*, 10(1-2):145–169, 2001.
16. O. Lassila and S. Dixit. Interleaving discovery and composition for simpleworkflows. In *Semantic Web Services, 2004 AAAI Spring Symposium Series*, 2004.
17. L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th International Conference on the World Wide Web*, 2003.

18. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.
19. S. McIlraith, T. Son, and H. Zeng. Mobilizing the semantic web with daml-enabled web services. In *Proc. Second International Workshop on the Semantic Web (SemWeb-2001)*, Hongkong, 2001.
20. 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.
21. M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proceedings of the 1st International Semantic Web Conference (ISWC)*, 2002.
22. S. R. Ponnekanti and A. Fox. Sword: A developer toolkit for web service composition. In *In 11th World Wide Web Conference (Web Engineering Track)*, 2002.
23. Sun Microsystems, Inc. Java HotSpot Technology. Web pages at `http://java.sun.com/products/hotspot/`.
24. K. Sycara, J. Lu, M. Klusch, and S. Widoff. Matchmaking among heterogeneous agents on the internet. In *Proceedings of the 1999 AAAI Spring Symposium on Intelligent Agents in Cyberspace*, Stanford University, USA, March 1999.
25. S. Thakkar, C. A. Knoblock, J. L. Ambite, and C. Shahabi. Dynamically composing web services from on-line sources. In *Proceeding of the AAAI-2002 Workshop on Intelligent Service Integration*, pages 1–7, Edmonton, Alberta, Canada, July 2002.
26. UDDI. Universal Description, Discovery and Integration Web Site, http://www.uddi.org/.
27. J. Vitek, C. Bryce, and W. Binder. Designing JavaSeal or how to make Java safe for agents. Technical report, University of Geneva, July 1998.
28. W3C. OWL web ontology language 1.0 reference, http://www.w3.org/tr/owl-ref/.
29. W3C. Web services description language (wsdl) version 1.2, http://www.w3.org/tr/wsdl12.
30. W3C. XML Schema, http://www.w3.org/xml/schema.
31. D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S web services composition using SHOP2. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, 2003.