An Extensible Directory Enabling Efficient Semantic Web Service Integration

Ion Constantinescu, Walter Binder, and Boi Faltings

Artificial Intelligence Laboratory
Swiss Federal Institute of Technology,
IN (Ecublens), CH–1015 Lausanne, Switzerland.
{ion.constantinescu,walter.binder,boi.faltings}@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 service integration. The directory implementation relies on a compact numerical encoding of service parameters and on a multidimensional index structure. It supports isolated service integration sessions providing a consistent view of the directory data. During a session a client may issue multiple queries to the directory and retrieve the results incrementally. In order to optimize the interaction of the directory with different service composition algorithms, the directory supports custom ranking functions that are dynamically installed with the aid of mobile code. The 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 (e.g., denial-of-service attacks). With the aid of user-defined ranking functions, application-specific ordering heuristics can be deployed directly. Experiments on randomly generated problems show that they 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, service directories, service discovery, service ranking.

1 Introduction

Service composition is an exciting area which has received a significant amount of interest in the last period. Initial approaches to web service composition [28] used a simple forward chaining technique which can result in the discovery of large numbers of services. There is a good body of work which tries to address the service composition problem by applying planning techniques based either on theorem proving (e.g., Golog [21,22], SWORD [24]) or on hierarchical task planning (e.g., SHOP-2 [32]).

[1] 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).

S.A. McIlraith et al. (Eds.): ISWC 2004, LNCS 3298, pp. 605–619, 2004.

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 [19] 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 [12]. 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 [12].

We have developed a directory service with specific features to ease and make more efficient service composition but without fully solving it. Queries may not only search for complete matches, but may also retrieve *partially matching* directory entries [10]. 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. 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 [9].

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 ranking functions*.

Custom 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 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 service integration sessions and user-defined ranking functions. We present some parts of the session API and explain the restricted programming model for ranking functions. Moreover, we explain how our directory protects itself against malicious code. Performance evaluations point up the necessity to support heuristic 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 discuss extensions of the directory specific to service composition that enable consistent views of the directory data during the service integration process. In Section 4 we show how the directory can be dynamically extended by user-defined ranking functions. We discuss some implementation details, in particular showing how the directory protects itself against malicious code. In Section 5 we present some experimental results that illustrate the need for dynamically installing application-specific heuristic ranking functions. Finally, Section 6 concludes this paper.

2 Service Composition with Directories

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.

Our composition algorithm builds on forward chaining, a technique well known for planning [6] and more recently for service integration [28]. Most previous work on service composition has required an explicit specification of the control flow between basic services in order to provide value-added services.

For instance, in the eFlow system [8], a composite service is modeled as a graph that defines the order of execution of different processes. The Self-Serv framework [2] uses a subset of statecharts to describe the control flow within a composite service. The Business Process Execution Language for Web Services (BPEL4WS) [7] addresses compositions where the control flow of the process and the bindings between services are known in advance. In service composition using Golog [22], logical inferencing techniques are applied to pre-defined plan templates.

More recently, planning techniques have been applied to the service integration problem [25,33,34]. Such an approach does not require a pre-defined process model of the composite service, but returns a possible control flow as a result. Other approaches to planning, such as planning as model checking [15], are being considered for web service composition and would allow more complex constructions such as loops.

Informally, the idea of composition with forward chaining is to iteratively apply a possible service S to a set of input parameters provided by a query Q (i.e., all inputs required by S have to be available). If applying S does not solve the problem (i.e., still not all the outputs required by the query Q are available) then a new query Q' can be computed from Q and S and the whole process is iterated. This part of our framework corresponds to the planning techniques currently used for service composition [28]. Details on our service composition algorithm, which also supports partial matches, can be found in [12].

2.1 Directories of Web Services

Currently, UDDI is the state of the art for directories of web services (see [9] for an overview of other service directory systems). The 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 [27], and the newer efforts geared towards OWL-S [23]. Other approaches include the Ariadne mediator [17].

Regarding service descriptions, in this paper we partially build on existing developments, such as [31], [1], and [13], by considering a simple table-based formalism where each service is described through 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). For efficiency reasons, we represent the DG numerically. Details concerning the encoding can be found in [10].

2.2 Match Types

We consider four match relations between a query Q and a service S (see example for input parameters in Fig. 1):

- **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.

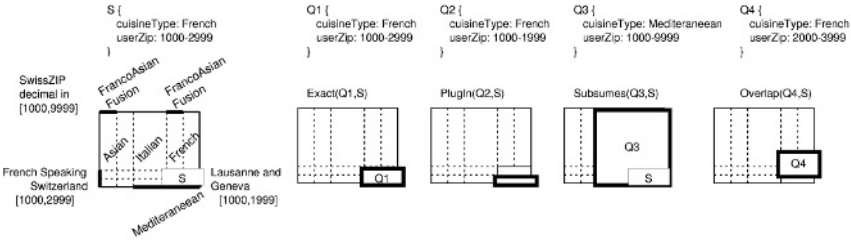

Fig. 1. Match types of inputs of query Q and service S by "precision": **Exact**, **PlugIn**, **Subsumes**, **Overlap**.

In the example we show how the match relation is determined between the inputs available from the queries Q1, Q2, Q3, Q4 and the inputs required by service S.

Given that a **Subsumes** match requires the specification of supplementary constraints we can order the types of match by "precision" as following: **Exact**, **PlugIn**, **Subsumes**, **Overlap**. We consider **Subsumes** and **Overlap** as "partial" matches. The first three relations have been previously identified by Paolucci in [23] and the forth, **Overlap**, was identified by Li [20] and Constantinescu [10].

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 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 [16]. 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.

3 Service Integration Sessions and Concurrency Control

As directory queries may retrieve large numbers of matching entries (especially when partial matches are taken into consideration), it is important to support incremental access to the results of a query in order to avoid wasting network bandwidth. Our directory service offers *sessions* which allow a user to issue queries to the directory and to retrieve the results one by one (or in chunks of limited size).

3.1 Session API

Fig. 2 presents the UML diagram of our directory service API. Sessions have a limited time-span after which session methods cannot be invoked anymore (timeout). The directory has per-session resource usage limits regarding the number of created ranking functions and the size of the result set of a query. The directory also checks for non-conforming ranking functions (see Section 4) and invalid service IDs.

The `Directory` interface is used to create a service integration session. Within a session several queries may be issued. The `Query` interface allows the sequential retrieval of query results, i.e., the directory entries that match the requirements of the query (see above Section 2.2 or [23]).

There is no particular order in which the results are returned (the order depends on the internal organization of the directory tree), unless a user-defined ranking function has been provided by the client (see Section 4). A session may define multiple ranking functions and associate a different ranking function with each query. Identifiers for declared ranking functions remain valid throughout a session, unless they are destroyed explicitly. Explicit destruction of ranking function IDs may be needed if the directory enforces a strict limit on the number of ranking functions active within a session. The ranking function is provided as bytecode of a compiled Java class.

The `getNextResult()` and `getNextResults()` methods do not return full service descriptions, but IDs that can be used later to retrieve the associated service description using the `getService()` method. The IDs remain valid and unique during the whole session. This approach allows the service integration client to cache retrieved service descriptions (the same service description may be returned in several queries) and thus helps to reduce the network traffic with the directory. The complete service description may be much larger than its ID of 64 bits.

Within a session several queries may be started in parallel. For each query the directory has to maintain some internal state. If there are too many concurrent queries in a session or a query requires too much processing or memory allocation within the

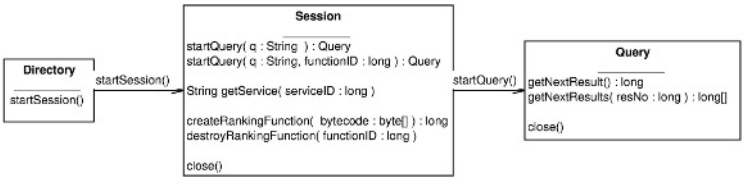

Fig. 2. The session API.

directory, an appropriate exception is thrown. When a session is closed, all its queries are closed automatically, too, and returned IDs of service descriptions may become invalid.

3.2 Session Implementation

The session guarantees a consistent view of the directory, i.e., the directory structure and contents as seen by a session does not change. Concurrent updates (service registration, update, and removal) do not affect the sequence of query results returned within a session; sessions are isolated from concurrent modifications.

Previous research work has addressed concurrency control in generalized search trees [18]. However, these concurrency control mechanisms only synchronize individual operations in the tree, whereas our directory supports long-lasting sessions during which certain parts of the tree structure must not be altered. This implies that insertion and deletion operations may not be performed concurrently with query sessions, as these operations may significantly change the structure of the tree (splitting or joining of nodes, balancing the tree, etc.).

The following assumptions underly the design of our concurrency control mechanism:

1. Read accesses (i.e., queries within sessions and the incremental retrieval of the results) will be much more frequent than updates.
2. High concurrency for read accesses (high number of concurrent query sessions).
3. Read accesses shall not be delayed.
4. Updates may become visible with a significant delay, but feedback concerning the update (success/failure) shall be returned immediately.
5. The duration of a session may be limited (timeout).

A simple solution would be to create a private copy of the result set of each query. However, as we want to support a high number of concurrent sessions, such an approach is inefficient, because it wastes memory and processing resources to copy the relevant data. Hence, such a solution may not scale well and is not in accord with our first two assumptions. Another solution would be to support transactions. However, this seems to be too heavy weight. The directory shall be optimized for the common case, i.e., for read accesses (first assumption). This distinguishes directory services from general-purpose databases. Concurrency protocols based on locking techniques are not in accord with these assumptions either. Because of assumption 3, sessions shall not have to wait for concurrent updates.

In order to meet the assumptions above, we have designed a mechanism which guarantees that sessions operate on read-only data structures that are not subject to changes. In our approach the in-memory structure of the directory tree (i.e., the directory index) is replicated up to 3 times, while the actual service descriptions are shared between the replicated trees.

When the directory service is started, the persistent representation of the directory tree is loaded into memory. This master copy of the directory tree is always kept up to date, i.e., updates are immediately applied to that master copy and are made persistent, too. Upon start of the directory service, a read-only copy of the in-memory master copy

is allocated. Sessions operate only on this read-only copy. Hence, session management is trivial, there are no synchronization needs. Periodically, the master copy is duplicated to create a new read-only copy. Afterwards, new sessions are redirected to the new read-only copy. The old read-only copy is freed when the last session operating on it completes (either by an explicit session termination by the client or by a timeout).

We require the session timeout to be smaller than the update frequency of the read-only copy (the duplication frequency of the master copy). This condition ensures that there will be at most 3 copies of the in-memory representation of the directory at the same time: The master where updates are immediately applied (but which is not yet visible to sessions), as well as the previous 2 read-only copies used for sessions. When a new read-only copy is created, the old copy will remain active until the last session operating on it terminates; this time span is bounded by the session timeout.

In our approach only updates to the master copy are synchronized. Updates are immediately applied to the master copy (yielding immediate feedback to the client requesting an update). Only during copying the directory is blocked for further updates. In accord with the third assumption, the creation of sessions requires no synchronization.

4 Custom Ranking Functions

Sessions allow a service integration engine to incrementally retrieve service descriptions from the directory that (partially) match some of the requirements of the query. The order in which matching service descriptions are returned depends on the actual structure of the directory search tree. 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 ranking and sorting according to application-dependent heuristics should occur directly within the directory. As for each service integration algorithm a different order may be advantageous, our directory allows its clients to define custom ranking functions which are used to sort the results of a query. This approach can be seen as a form of remote evaluation [14].

4.1 API for Ranking Functions

The ranking function receives as arguments information concerning the matching of a service description in the directory with the current query. It returns 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 ranking function (a higher value means a better match). Results for which the ranking function evaluates to zero come at the end, a negative value for the ranking function indicates that the match is too poor to be returned, i.e., the result is discarded and not passed to the client.

As arguments the client-defined ranking function takes 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 below Fig. 3). The size and membership methods have as single argument the current `ParamSet` object while the union, intersection and difference methods use two arguments - the current object and a second `ParamSet` object passed as parameter.

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

1. basic **set operations**, where a set member is defined through a parameter name and it's 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 it's type in the resulting set is computed using 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` - this method returns true when the current set contains a parameter with the same name as that passed as parameter regardless of it's type.
- `union` - this method returns all parameters in the two argument sets. For any parameter that is common to the two argument sets the type in the resulting set is computed accordingly to the user-specified expression `newTypeExpr`.
- `intersection` - this method 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`. Those parameters are added to the resulting set, with a type computed accordingly to the user-specified expression `newTypeExpr`.
- `minus` - this method returns the parameters that are either present only in the `ParamSet` object on which the method is called or 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 accordingly to the user-specified expression `newTypeExpr`.

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 arguments of a `union`, `intersection` or `minus` method. For such kind of parameter we note it's type in the two argument sets as `A` and `B`. The expressions are created from these two types possibly by using some extra constructors based on the Description Logic language OWL [30] like $\top$, $\bot$, $\neg$, $\sqcap$, $\sqcup$, $\sqsubseteq$, $\equiv$. The expressions are build by specifying a constructor type and which of the argument types `A` and `B` should be considered 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 constructor type A only $\neg A$ can be set).

We represent an expression as a bit vector having a value corresponding to it's 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

Table 1. Possible expressions for constructors of parameter sets.

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	$A \equiv B, A \equiv \neg B, \neg A \equiv B, \neg A \equiv \neg B$

```
public interface Ranking {
  double rank( 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.

NEG_B. For the actual encoding see Table 1. 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 two sets X and Y, which have in X a type that is more specific than the one in Y. We would like to preserve in the result set the type values in X. The following statement can be used for this purpose: `X.intersection(Y, ParamSet.SUPERCLASS, ParamSet.A)`.

The directory supports 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.

On one hand, this API allows to write rather generic ranking functions that take into account only the number of parameters in a certain matching relationship between the description of the query and the description of the service. On the other hand, the ranking function may be customized for a query for which the existance of particular parameters is important by using the set membership test (e.g., if a required output is known to be very hard to be found, a service description that provides exactly this output may receive a higher ranking).

4.2 Exemplary Ranking Functions

In the example in Fig. 4 two basic ranking functions are shown, the first one more appropriate for a service composition algorithms using forward chaining, the second for algorithms using backward chaining with complete type matches.

```
public final class ForwardRanking implements Ranking {
 public double rank( ParamSet qin, ParamSet qout,
                     ParamSet sin, ParamSet sout ) {
  // 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 final class BackwardCompleteRanking implements Ranking {
 public double rank( ParamSet qin, ParamSet qout,
                      ParamSet sin, ParamSet sout ) {
   // 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());
 }
}
```

Fig. 4. Exemplary ranking functions.

4.3 Safe and Efficient Execution of Ranking Functions

Using a subset of Java as programming language for 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 [26] 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 ranking functions are executed as efficiently as algorithms directly built into the directory.

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 [29] and in the J-SEAL2 [3,4] mobile object kernels. Our bytecode verifier ensures the following conditions:

- The `Ranking` interface is implemented.
- Only a single method is provided.
- The control-flow graph of the `rank()` method is acyclic. The control-flow graph

is 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 [5]). 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 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 ranking function is bounded by the size of its code. Hence, an attacker cannot crash the directory by providing, for example, a ranking function that contains an endless loop. Moreover, ranking functions cannot allocate memory. Our extended bytecode verification algorithm is highly efficient, its performance is linear with the size of the ranking method. As a prevention against denial-of-service attacks, our directory service allows to set a limit for the size of ranking functions.

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.

Service composition sessions that do not define custom ranking functions require only a small amount of memory within the directory in order to keep track of the traversal of the directory tree. However, when custom ranking functions are used, the directory has to allocate a temporary data structure in order to sort the matching entries according to the ranking function. In order to protect the directory from attacks, queries issued in sessions that employ custom ranking functions are rejected if the size of the result set exceeds a certain threshold defined by the directory service provider. In such a case, the

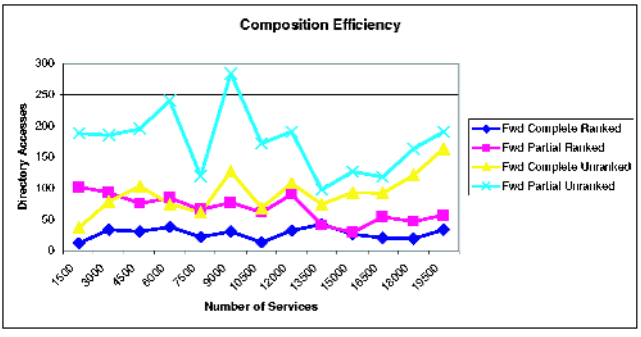

Fig. 5. Impact of ranking functions on composition algorithms.

client has either to do without the custom ranking function, or the query has to be made more specific in order to reduce the number of results.

5 Evaluation

We have evaluated our approach by carrying out tests on random service descriptions and service composition problems that were generated as described in [11]. As we consider 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 that can compose partially matching services, too [12]. 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 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 ranking functions. 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 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).

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 and a consistent view of the directory data, as well as user-defined 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 ranking heuristics. Thanks to the custom 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. As user-defined ranking functions may be abused to launch denial-of-service attacks 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. B. Benatallah, Q. Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, 2003.
3. 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.
4. 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.
5. 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.
6. A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300, 1997.
7. BPEL4WS. Business process execution language for web services version 1.1, http://www.ibm.com/developerworks/library/ws-bpel/.
8. F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and dynamic service composition in eflow. Technical Report HPL-2000-39, Hewlett Packard Laboratories, 2000.
9. 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.
10. I. Constantinescu and B. Faltings. Efficient matchmaking and directory services. In *The 2003 IEEE/WIC International Conference on Web Intelligence*, 2003.
11. 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.
12. 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.
13. DAML-S. DAML Services, http://www.daml.org/services.
14. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

15. F. Giunchiglia and P. Traverso. Planning as model checking. In *European Conference on Planning*, pages 1–20, 1999.
16. 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.
17. 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.
18. M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and recovery in generalized search trees. In J. M. Peckman, editor, *Proceedings, ACM SIGMOD International Conference on Management of Data: SIGMOD 1997: May 13–15, 1997, Tucson, Arizona, USA*, 1997.
19. O. Lassila and S. Dixit. Interleaving discovery and composition for simpleworkflows. In *Semantic Web Services, 2004 AAAI Spring Symposium Series*, 2004.
20. 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.
21. 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.
22. 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.
23. 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.
24. 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.
25. B. Srivastav. Automatic web services composition using planning. In *International Conference on Knowledge Based Computer Systems (KBCS-2002)*, 2002.
26. Sun Microsystems, Inc. Java HotSpot Technology.
Web pages at `http://java.sun.com/products/hotspot/`.
27. 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.
28. 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.
29. J. Vitek, C. Bryce, and W. Binder. Designing JavaSeal or how to make Java safe for agents. Technical report, University of Geneva, July 1998.
30. W3C. OWL web ontology language 1.0 reference, http://www.w3.org/tr/owl-ref/.
31. W3C. Web services description language (wsdl) version 1.2, http://www.w3.org/tr/wsdl12.
32. 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.
33. Wu, Dan and Parsia, Bijan and Sirin, Evren and Hendler, James and Nau, Dana. Automating DAML-S Web Services Composition Using SHOP2. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, 2003.
34. J. Yang and M. P. Papazoglou. Web component: A substrate for Web service reuse and composition. *Lecture Notes in Computer Science*, 2348, 2002.