Service Composition with Directories

Ion Constantinescu, Walter Binder, and Boi Faltings

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

Abstract. This paper presents planning-based service composition algorithms that dynamically interact with a potentially large-scale directory of service advertisements in order to retrieve matching service advertisements on demand. We start with a simple algorithm for untyped services, similar to a STRIPS planner. This algorithm is refined in two steps, first to exploit type information, and second to support partial type matches. An evaluation confirms that the algorithms scale well with increasing size of the directory and that the support for partial type matches is essential to achieve a low failure rate.[1]

Keywords: Service composition, planning, service discovery, semantic web services.

1 Introduction

Today the predominant way we interact with the Web is via browsers that manipulate information by rendering it in a human-readable way. However, there is an evolution towards the automatisation of many processes on the Web, which may result in computer-to-computer interactions becoming predominant over current human-to-computer interactions. For modeling computer-to-computer interactions, currently the de-facto paradigm is that of "services". But making services automatically interact with each other raises a number of difficult problems, currently under hard scrutiny by industrial and academic research.

The Semantic Web [14] is fundamental for such computer-to-computer interactions to become reality, since it provides an universally accessible platform and computer-understandable semantics for data to be shared and processed by automated tools. Experts have already developed a range of mark-up frameworks and languages, notably the revised Resource Description Framework (RDF) [29] and the Web Ontology Language (OWL) [28], which mark the emergence of the Semantic Web as a broad-based, commercial-grade platform.

Service discovery is the process of locating providers advertising services that can satisfy a service request specified by a service consumer. *Automated service composition* addresses the problem of assembling services based on their functional specifications in order to achieve a given task and to provide extra functionality. When discovery

[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).

W. Löwe and M. Südholt (Eds.): SC 2006, LNCS 4089, pp. 163–177, 2006.

fails in locating a single service, service composition can be used instead for satisfying a service request. Thus, service composition may be regarded as a generalized form of discovery. So far, most approaches that have been suggested for automated service composition are based on planning techniques.

Due to the open and changing environment in which it is performed, service discovery needs to operate without any specific prior knowledge of existing services. Services are therefore indexed in directories, and the main goals for the implementation of these directories and the matchmaking algorithms are to maximize the success rate as well as the efficiency of processing queries.

Therefore, planning algorithms used for automated service composition have to be adapted to a situation where operators are not known a priori, but have to be retrieved through queries to these directories. Basic planning systems check all operators in the planning library against the current search state in order to determine which action to perform next. In contrast, in the case of service composition, the search state is used to extract the *specification of possible operators*. This specification together with some constraints specific to the composition algorithm is used to formulate a query to the service directory.

The original contribution of this paper is the presentation of service composition algorithms that dynamically interact with a directory to retrieve relevant service descriptions. Our approach to automated service composition is based on matching input and output parameters and world states prior and posterior to the invocation of a service in order to constrain the ways in which services may be composed. In this paper we show the stepwise refinement of a simple planning algorithm in order to take the types of input resp. output parameters into account and to support also the composition of services with *partially matching parameter types*, which significantly reduces the failure rate of the composition algorithm.

The approach presented here builds on our research on service directories [8,3,9] and refines and extends our previous work on service composition [12] by supporting a more expressive service description formalism and detailing the interaction of our service composition algorithms with a directory. Moreover, this paper includes the first detailed presentation of our composition algorithms, which we derive from a standard planning approach. New measurements of our approach complete this paper.

The rest of this paper is structured as follows: The next section discusses some related work in the area of service composition. In Section 3 we explain how we represent service advertisements and service requests. Section 4 discusses requirements and gives an overview of our service composition engine. Section 5 introduces a simple composition algorithm, which does not take type information into account. It is the basis for the subsequent refinements in Section 6 and Section 7. In Section 8 we evaluate performance and failure rate of the presented service composition algorithms. Finally, Section 9 concludes this paper.

2 Related Work

For the last years, service integration has been an active field in both the AI and database research communities, including Infosleuth [1] and work by Doan and Halevy [13].

Another approach is that of Thakkar and Knoblock [26], concretized by the Building Finder application, where a number of manually defined data-sources, such as the Microsoft Terraservice, U.S. Census Bureau information files, as well as geocoding information and different real estate property tax sites, where composed using a forward chaining technique.

Some approaches to composition require an explicit specification of the control flow between basic services in order to provide value-added services. For instance, in the eFlow system [6], 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) [5] addresses compositions where the control flow of the process and the bindings between services are known in advance.

There is a good body of work which tries to address the service composition problem by using planning techniques based either on theorem proving (e.g., ConGolog [20,21] and SWORD [25]) or on hierarchical task planning (e.g., SHOP-2 [32]). Such approaches do not require a pre-defined process model of the composite service and return a possible control flow as result. In the scenario used by the ConGolog approach, the composition engine would have to book flight tickets and arrange ground transportation and hotel reservations. For SWORD, the example used was of a composite service giving driving directions to one's home. The composite service was formed from two services, one that mapped names to addresses and another that was giving driving directions to a given address. In the motivating example in the SHOP-2 approach, for handling a medical emergency, several data sources had to be composed and a schedule had to be computed.

Other approaches based on planning, such as planning as model checking [16], are being considered for Web service composition and would allow more complex constructs such as loops [27].

The main drawback of all previously presented approaches to service composition based on planning is that they assume that relevant service descriptions are initially loaded into the reasoning engine and that no service discovery is performed during composition. I.e., these approaches do not fit an environment where a large number of dynamically changing service advertisements are published in service directories. In contrast, the service composition algorithms presented in this paper dynamically retrieve relevant service advertisements from a potentially large-scale service directory.

3 Service Descriptions

A service description specifies aspects related to the functionality available from a service provider (service advertisement) or requested by a service consumer (service request). We represent service advertisements and service requests through variables and constraints on these variables. Variables refer to required or provided service parameters or to aspects related to the state of the world before or after the invocation of the service. Constraints specify the possible combinations of values that different variables can take.

In our formalism, each variable is defined by two elements:

- A *description* specifying the actual semantics of the data the variable is holding, including information whether the variable represents a parameter or a world state (e.g., in a travel domain the description of a variable could be $DepartureParam$ or $ArrivalParam$ and the description of a world state could be $ValidCreditState$). Usually, the description is directly associated to the name of the variable itself.
- A *type* defining the way data of the variable is represented and the set of values that the variable can take (e.g., possible values for $DepartureParam$ and $ArrivalParam$ could be represented by the sets $\{Geneva, Basel\}$ and $\{Barcelona, Nice\}$, where all four city values could be of type $Location$).

We presume that both variable *descriptions* and *types* can be defined using a class/ontological language like OWL [28]. Primitive data-types used for specifying the variable *type* can be defined using a language like XSD [30].[2]

Constraints on variables can specify either *preconditions* (set of possible parameter and world state values required to be true prior to the invocation of the service) or *effects* (how parameters and world states are affected by the execution of the service). Constraints are specified in the form of sets of possible variable assignments. Each assignment represents a set of variable/value pairs. Constraints are identified by keywords (e.g., PRE for preconditions, respectively EFF for effects).

We call parameter or world state variables appearing in PRE constraints $prior$ variables, and parameter or world state variables appearing in EFF constraints $post$ variables.

In a service advertisement, variables and constraints describing parameters and world states have the following semantics:

- In order for the service to be invokable, a value must be known for each of the *prior* variables and it has to be consistent with the respective semantic *description* and syntactic *type* of the variable. The value provided as $prior$ parameter or world state has to be semantically at least as specific as what the service is able to accept. Regarding the variable type, in the case of primitive data types the invocation value must be in the range of allowed values, or in the case of classes the invocation value must be subsumed by the type of the variable.
- Upon successful invocation, the service returns a value for each of the $post$ parameters, and the execution engine assigns a value to each of the variables representing $post$ world states. Each of these values is consistent with the respective *description* and *type* of the variable.
- Regarding preconditions, in order for the service to be invokable, at least one assignment set in the constraint has to be satisfied by the current values of variables defining parameters and states of the world.
- Effect constraints represent guarantees on the possible combinations of values for variables describing $post$ parameters and world states as well as how $prior$ world states are maintained after the invocation of the service (e.g., the effect of an action modeling a robot picking up a block from a table will not maintain the fact that the block is on the table, which is part of the action's preconditions).

[2] At the implementation level both primitive data-types and classes are represented as sets of numeric intervals [8].

Service requests are represented in a similar manner but have different semantics:

- The service request's $prior$ variables describe available parameters (e.g., provided by the user or by another service) or aspects of the world specifying an initial state of facts. Each of these variables has attached a semantic *description* and either a *type* or a concrete *value*.
- The service request's $post$ variables represent parameters that a compatible (composite) service must provide and world states that specify aspects of the state of the world that have to be influenced by the execution of the service. The variable description defines the actual semantics of the required information and the variable type defines what ranges of values can be handled by the requester. A compatible (matching) service must be able to provide a value for each of the $post$ parameters and world states of the service request, semantically at least as specific as the requested variable description, and having values in the range defined by the requested parameter type.
- Preconditions in a service request represent restrictions on the allowed combinations of values for available parameters or initial world states described by *prior* variables.
- Effects represent restrictions on the allowed combinations of values for variables describing required *post* parameters or world states that the service request is willing to accept.

We use the following functions to access the variables of service advertisements and service requests:

- $vars(prior \mid post, S)$ – Returns the set of $prior$ or $post$ variables for a service description S. We assume variables to be described as concepts using a language like OWL [28], conforming with the semantics for the Description Logic operators $\equiv, \sqsubseteq, \sqcap, \sqcup, \bot, \top$. As previously specified, the description of each variable specifies if it is a parameter or a world state.
- $type(V, S)$ – Returns the type of the variable named V in the frame of a service description S as the set of possible values that V can take. The $\equiv$, $\subseteq$ and $\cap \neq \emptyset$ operators in conjunction with this function can be used to determine if two value sets are equivalent, subsume each other, or are overlapping.
- $constraint(PRE \mid EFF, S)$ – Returns the set of possible variable assignments (variable/value pairs) for the precondition or effect constraints in the service description S.

4 Requirements and High-Level Architecture

Classic approaches to planning assume domains with a relatively small number of operators (e.g., domains for the International Planning Competition [17] have some dozens actions). For solving planning problems in these kinds of domains, the difficulties that need to be addressed are related to the large space of possible states to be searched and to the embedded hard resource-allocation problems.

In this paper, we are concerned by the following issues, which are specific and unique to the composition of services deployed in open environments:

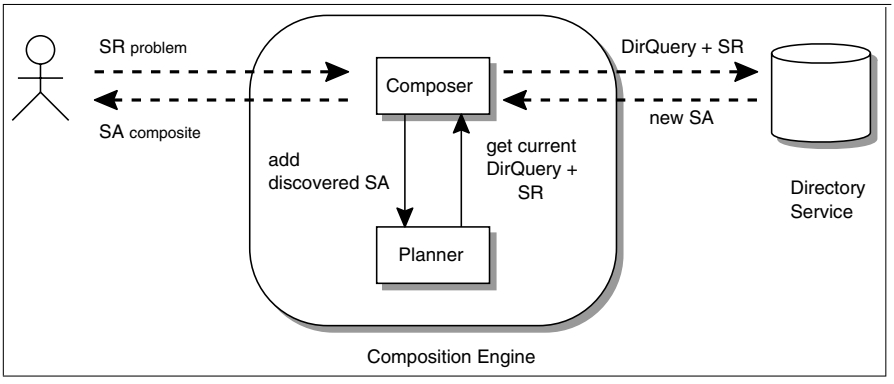

Fig. 1. Interactions of service composition engine using a directory

- **Discovery in large-scale directories** – We assume that a large number of service advertisements is stored in (possibly distributed) yellow-page directories. How can we discover exactly the services that are relevant at each step of the composition process?
- **Runtime non-determinism** – When discovered services match only partially but not completely[3], the reasoning engine has to aggregate several services as switches in order to fulfill the required functionality. The actual flow of messages will be routed based on runtime values on the appropriate paths. How can we discover and create those switches and how can we make sure that they correctly handle all possible combinations of parameter values?

The design choices that we took in our approach to service composition are driven by the above requirements and based on the following assumptions:

- **Large result sets** – For each query, the service directory may return a large number of service advertisements.
- **Costly directory accesses** – Being a shared resource, accessing the directory (possibly remotely) is expensive.

The design of our service-composition engine addresses these issues by interleaving discovery and composition and by computing the "right" query at each step. Hence, the architecture of our service composition system consists of three separate components (see Fig. 1):

- **Planner** – A component that is able to compose one or more known service advertisements (SAs) in order to satisfy an initial service request $SR_{problem}$. The composition is formulated as a service advertisement $SA_{composite}$, represented as a workflow. If using the currently known services the planner is unable to generate

[3] We consider as partial matches the *subsume* match type identified by Paolluci [23] and the *intersection* or *overlap* match type identified by Li [19] and Constantinescu [10].

Algorithm 1. Composer algorithm for service composition with directories

```
procedure FindComposition(PL, DIR, SR_problem) : SA_composite
    PL.init(SR_problem)
    SR ← SR_problem
    while (SA ← DIR.nextResult(SR, PL.getDirQuery())) ≠ ∅ do
        PL.addService(SA)
        if PL.isFeasible() ∧ (SA_composite ← PL.solve()) ≠ ∅ then
            return SA_composite
        SR ← PL.getServiceRequest()
```

such a composition, the state of the current composition search can be used to extract a new service request SR. Together with a planner-specific $DirQuery$, a new directory query can be formed. See [3,9] for details concerning our directory query language that is used to specify a $DirQuery$.

- **Service directory** – Stores service advertisements and is able to process queries formulated as a pair of a service request SR and a $DirQuery$. The query result consists of one or more service advertisements SA that can be incrementally returned to the composition engine.
- **Composer** – A component that implements the interleave between planning and discovery by using the current search state of the planner to generate new service requests SRs which are used for the discovery of additional service advertisements SAs. These advertisements are then added to the planner either until the initial service request $SR_{problem}$ is satisfied or no more results can be found.

In Algorithm 1 we present our approach to service composition with directories. As outlined before, the algorithm makes use of a planner PL and directory DIR in order to solve a composition problem formulated by the service request $SR_{problem}$. If successful, the system returns a composition in the form of a service advertisement $SA_{composite}$, represented as a workflow.

The algorithm relies on the following functions:

- $PL.init(SR)$ – Initializes the planner according to a given service request $SR_{problem}$.
- $DIR.nextResult(SR, DirQuery) : SA$ – Queries the directory for advertisements consistent with the given SR according to the matching relation defined by $DirQuery$. The best ranking SA accordingly to $DirQuery$ is then returned. For details concerning the ranking of matching service advertisements, see [3,9].
- $PL.addService(SA)$ – Adds a newly discovered service advertisements SA to the planner, possibly updating search structures internal to the planner.
- $PL.getDirQuery() : DirQuery$ – Returns a planner-specific $DirQuery$ that together with a service request SR can be used for formulating a query to the service directory.
- $PL.getServiceRequest() : SR$ – Returns a service request that reflects the current state of the search for composition.

Algorithm 2. Initialization and query generation functions of a planner for untyped deterministic services

procedure $PL.init(SR)$
  $PL.states(0) \leftarrow constraint(PRE, SR)$
  $lastLevel \leftarrow 0$
  $PL.goals \leftarrow constraint(EFF, SR)$
procedure $PL.getServiceRequest() : SR$
  $SR \leftarrow newRequest()$
  $constraint(PRE, SR) \leftarrow PL.states(PL.lastLevel)$
  $constraint(EFF, SR) \leftarrow PL.goals$
  return SR

- $PL.isFeasible() : Boolean$ – This is a computationally cheap test that represents a necessary condition for the plan to be solvable using the current service advertisements. If this function returns true, the search for a plan might be successful. If it returns false, the search for a plan will certainly fail (i.e., further services have to be discovered from the directory).
- $PL.solve() : SA$ – When the plan is feasible, this function uses the currently known service advertisements to search for a composition that fulfills $SR_{problem}$.

In the next sections we present in detail several planning systems of different complexity. The first one is able to handle untyped deterministic services, the second one handles compositions of completely matching typed services (deterministic compositions), and finally the last one allows to compose typed services that partially match, where the non-determinism in the composition is addressed by the creation of *switches* in the resulting workflow.

5 Composing Untyped Deterministic Services

By untyped services we understand service descriptions (service advertisements and requests) for which all possible values in specification assignments are disjoint (e.g., for data-types their intervals must not overlap, and any two concepts must satisfy the $\cap = \perp$ relation).

Moreover, we assume that service advertisements are completely deterministic: their precondition PRE and effect EFF constraints each contain exactly one assignment set. The same applies to the precondition constraints in the service request $SR_{problem}$. We allow precondition constraints (available states) in non-problem requests (like those generated by the planner) to contain several assignments. Effect constraints (required goals) in requests can always contain several assignments.

As assignment sets with disjoint values can be seen as sets of propositions, untyped deterministic services are equivalent with actions and problems of the STRIPS planning formalism [15]. As such, the structure of our planner is similar to the structure of well-known STRIPS planners like Graphplan [4].

Internally, our planner maintains a structure of alternating action and state levels. A state level n accessible by $PL.states(n)$ represents the set of states reachable from the

initial state by the application of n services. In contrast to Graphplan, we assume that only one service can be applied at each level. The planner for untyped deterministic services uses the variable $PL.lastLevel$ for holding the current number of levels and a special level (set of states) for holding the problem goals $PL.goals$.

For this planner, $PL.init(SR)$ (see Algorithm 2) uses the preconditions of the problem request SR to initialize the first level and the problem effects to initialize the goal states. $PL.getServiceRequest()$ returns a new request (created by the $PL.newRequest()$ function) for which the precondition constraint is taken from the last level and the effect constraint from the goals.

The $DirQuery$ returned by this planner's $PL.getDirQuery()$ method ensures that all inputs required by a matching service advertisement are provided by the service request SR. As all services are assumed to be untyped, the $DirQuery$ ignores the matching of types. Moreover, the assignment set of the service advertisement's precondition constraint has to be included in SR's assignment set (potentially available states). The results are sorted so that matching service advertisements providing more of the outputs or goals required by SR come first. Concrete examples of similar directory queries can be found in [3,9].

The main complexity of the planner stems in the methods for addition of services and for solving the composition problem by searching the structure of levels. The $PL.addService(SA_{new})$ procedure has three phases: First, it tries to find the level at which a newly discovered service SA_{new} should be added. Next, it propagates new states generated by the addition of the service or the application of existing services over the states stored by current levels. Finally, a fixpoint procedure is used to extend the current levels by new ones (as long as the last levels are not the same).

$PL.addService(SA)$ used two additional functions: $PL.canApply(states, SA)$ tests whether a service advertisement SA can be added to a given set of states (level), and $PL.apply(states, SA)$ computes the effect states that result from applying a service advertisement SA to the set of states. These two functions operate under the following assumptions, specific to untyped deterministic services:

- Deterministic SAs – Each service advertisement has exactly one set of assignments in its precondition and effect constraints.
- Explicit maintenance of preconditions – Variable/value tuples present in the precondition constraint of an advertisement that *do not* appear in the effect constraint are removed (delete list).

$PL.isFeasible()$ returns true if the last level contains some of the possible sets of goals.

To search for a planning solution, the $PL.solve()$ function recurses over the levels of the plan, starting from the initial level. If a sequence of services marking a solution has been found, a new composite service advertisement is created and returned. The composite service advertisement is a sequential workflow without any branches.

$PL.solve()$ maintains the state that can be achieved by currently selected services and at each step performs the following tasks: First, it checks whether a solution has been reached and if so it returns the solution. Otherwise, if the current state can be further expanded, it applies all services on the level to the current state and recursively checks for a solution. Finally, if no solution has been reached, neither by the current

state nor by the recursive call, $PL.solve()$ fails. A solution is discovered if the goal states are included in the current search state.

6 Composing Completely Matching Typed Services

Typed services correspond to non-deterministic planning operators, such as e.g. those supported by the ADL language [24]. Our approach for composing this kind of services is similar to the approach of Kuter and Nau [18], in which deterministic planning algorithms are enhanced in order to support non-determinism. This relies on the fact that conceptually, deterministic planning can be seen as search in a state space, whereas non-deterministic planning can be seen as a very similar search in a belief space, where a belief is represented as a set of states. Still, a major difference between non-deterministic planning and our approach is that the process of composing typed services does not support negative effects.

Next, we present the updates that our previous procedures need to undergo in order to be able to handle sets of states (beliefs) instead of simple states.

Concerning service descriptions, we do not maintain any restriction: values of assignments can be overlapping and constraints can have several assignments. Still, we assume that initial service descriptions (service request and service advertisements retrieved from the directory) are discretized, which may result in some constraints possibly containing extra assignments. For example, the constraint $\{< A, [10 - 20] >, < B, [0 - 10] >\}$ might be discretized along the A variable by the value 15 and along the B variable by the value 5 resulting in four constraints, equivalent with the initial one: $\{< A, [10 - 15] >, < B, [0 - 5] >\}, \{< A, [15 - 20] >, < B, [0 - 5] >\}$, $\{< A, [10 - 15] >, < B, [5 - 10] >\}, \{< A, [15 - 20] >, < B, [5 - 10] >\}$. As a result, in the composer all values are disjoint and as for untyped services we can consider assignments as sets of propositions, where a proposition maps to a variable/value pair.

We call the current planner "complete matching" due to the procedure for selecting services while searching for a plan. Even though we allow for a service to produce non-deterministic effects, the current planner will create a solution where the precondition constraints of services applied at each step (the set of states for which the service can be invoked) completely match or subsume the set of possible states available until that planning step, either from the initial conditions or generated by services applied so far. This implies that applying the service at that step will always work, even in the presence of non-determinism. These kinds of plans are also described in reference [7] and are called *strong solutions*.

In the case of our non-deterministic planner, the initialization and query extraction procedures $PL.init(SR)$ and $PL.getServiceRequest()$ are the same as for the untyped planner.

The $DirQuery$ returned by $PL.getDirQuery()$ is updated such that service advertisements matching any possible state are selected. It takes types into account, which have to be at least as general in the service advertisement as in the service request SR. For precondition constraints, we require that an overlap exists between the states accepted by the service advertisement and the ones provided by SR.

As for the previous planner, the current one maintains a Graphplan-like structure of levels containing states and services. Consequently, the only difference in

the procedure of adding a newly discovered service to the current structure stems in the fact that preconditions and effects might include multiple sets of states. This is reflected only in the procedures for testing the viability of applying a service $PL.canApply(states, SA)$ and actually computing the states resulting by the application $PL.apply(states, SA)$. Since a level contains the union of effects of different possible plans, the $PL.canApply(states, SA)$ function considers a service to be applicable only to level states that are supersets of states in the precondition of the service advertisement.

We use the same $PL.isFeasible()$ function as for the deterministic planner before; it returns true if some of the goal sets can be satisfied by the currently reachable states.

The main search function $PL.solve()$ is updated to deal with sets of states instead of a single state. The important difference between the current $PL.solve()$ function and the one for deterministic domains stems in the $PL.canApply(states, SA)$ function, which selects only services having the set of preconditions completely subsuming the current states of the search. In other words, we select services that will always work, no matter which of the current states will be true at runtime. The condition is similar to the one for determining the plan solution. Also the $PL.apply(states, SA)$ function is different from the one for untyped services in that no states are removed in this case (negative effects are not supported).

For determining when a given set of states represents a solution, we require that any of the possible states reachable by the current plan contains some goal states. This is equivalent with the notion of *strong solution* introduced for non-deterministic plans in reference [7].

7 Composing Partially Matching Services

Frequently, forward chaining with complete type matches is too restrictive and fails to find a solution, because the types accepted by the available service advertisements may partially overlap the type specified in a service request. For example, a service request for restaurant recommendation services across all Switzerland may specify that the integer parameter zip code could be in the range [1000,9999], whereas an existing service providing recommendations for the French speaking part of Switzerland accepts only integers in the range [1000-2999] for the zip code parameter. Nonetheless, there may be several recommendation services for different parts of Switzerland that together could cover the whole range given in the query. Hence, a service composition algorithm could create a workflow with different execution paths, depending on the concrete value provided for the zip code parameter at execution time. That is, the workflow would include a *switch* in order to select the appropriate execution path.

A novelty of our approach to composition stems in the capability of our planner to use service advertisements that partially match the current search states while the generated plans will still remain *strong solutions*. This kind of partial matching between service descriptions corresponds to the *overlap* or *intersection* match identified by Li [19] and Constantinescu [10]. In this respect, $PL.getDirQuery()$ returns a less restrictive $DirQuery$ than before, selecting service advertisements that have overlapping types with the service request [3,9].

Handling this kind of partial matches transforms our search into an AND/OR process: all states of the current belief have to lead to a solution (AND) while for each of them the process of searching for the right service advertisement(s) to be applied (OR) has to be recursively invoked.

Of course, exhaustively trying to recursively solve all states in a belief would have a tremendous impact on the performance of the search algorithm. To address this problem, the main idea of our approach relies on the fact that a belief is to be considered solved when all its states recursively lead to a solution. But for each of these states, once the fact that it leads to a solution has been established, it will remain so across different other searches and even in the case of adding new services.

Therefore, the approach that we take for enhancing our AND/OR search is to use a dynamic programming technique that globally marks the states of a belief successfully solved and re-uses this information for pruning OR branches at different invocation levels in the current search or in other further searches. It has to be noted that due to the nature of the search, we cannot anymore represent the planner solution as a sequential workflow, but have to create a tree structure in order to represent the switches.

8 Experimental Evaluation

We evaluated the service composition algorithms explained in Section 6 and Section 7 with our testbed presented in [11]. The testbed covers several application domains and

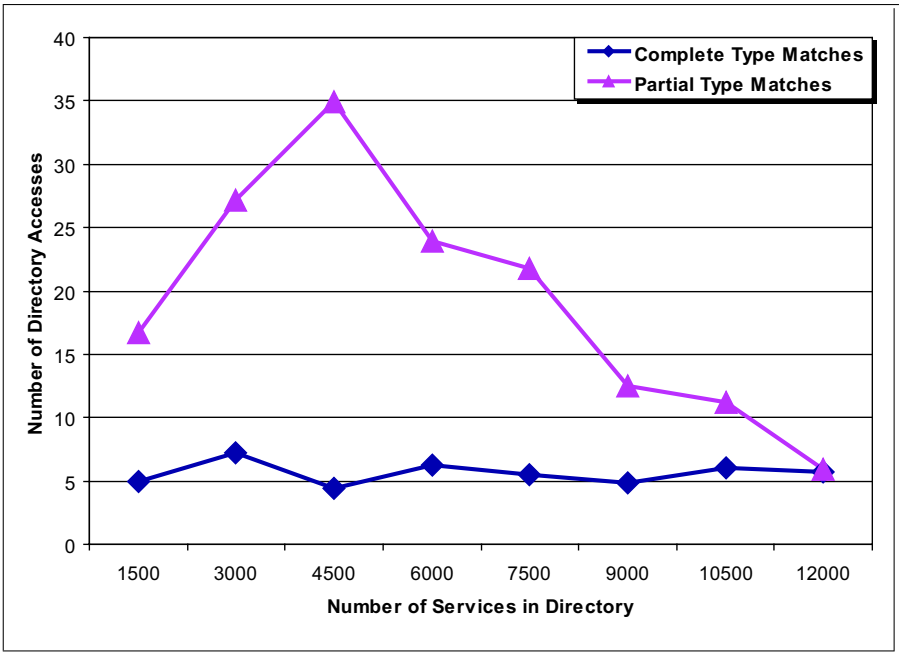

Fig. 2. Algorithm performance for successful compositions, measured in the number of directory accesses

allows to generate randomized service advertisements and service requests for these domains. We populated our directory [3,9] with an increasing number of generated service advertisements (1500–12000) and executed our service composition algorithms using the generated service requests as input. Each measurement represents the average of 50 runs with different service requests.

We used the number of directory accesses as performance metric. As it can be seen in Fig. 2, service composition with complete type matches scales well, because even in the presence of large numbers of service advertisements in the directory the number of required directory accesses does not increase significantly. Service composition with partial type matches is more costly concerning the number of directory accesses, in particular when composition problems are very hard (in our experiments when the directory contains about 4500 services). This is due to the fact that when the directory contains some services but not enough, failure of the composition cannot be determined easily since a good number of services are relevant but not enough of them can be composed to fully satisfy the given service request.

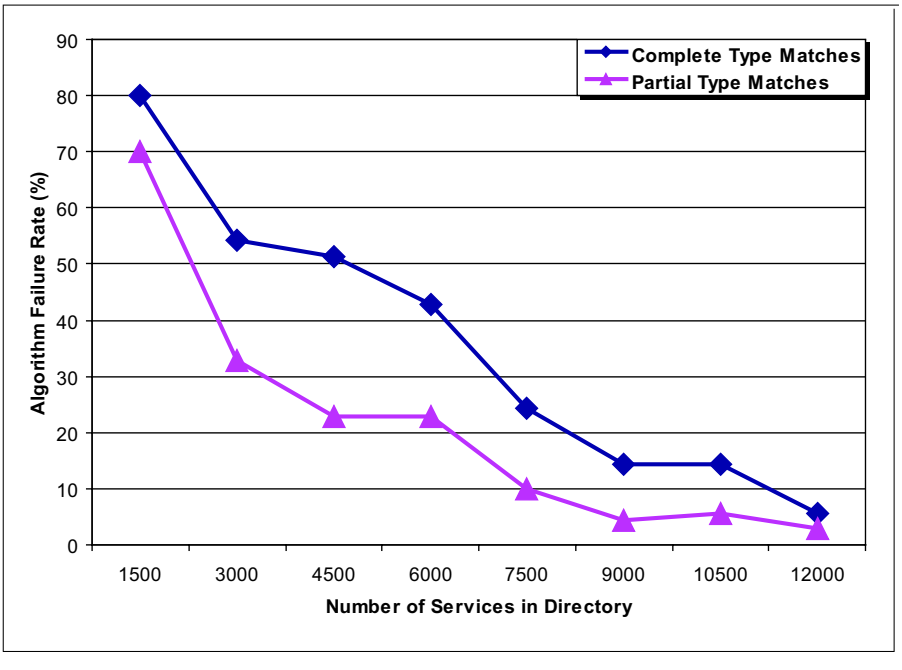

Fig. 3. Algorithm failure rate

As shown in Fig. 3, a major drawback of the service composition algorithm with complete type matches is the high failure rate of the composition, in particular when the directory does not contain too many services (up to 80% failure rate). Service composition with partial type matches significantly reduces the failure rate.

9 Conclusion

With the move towards Web services, tools for service composition are becoming increasingly important. Prevailing approaches to automated service composition, which are often straightforward adaptations of standard planning algorithms, require all service advertisements to be pre-loaded into the reasoning engine. Such techniques are not applicable in an open environment populated by a large number of dynamically changing services.

In this paper we presented planning-based service composition algorithms that dynamically access a separate, potentially large-scale directory in order to retrieve relevant service advertisements. We started with a simple planning algorithm for untyped services and refined it to first exploit type information, and second to support partial type matches. Experiments show that our algorithms scale well with the number of service advertisements stored in the directory. Moreover, the support for partial type matches brings significant gains in the number of problems that can be solved by automated service composition with a given set of service advertisements.

References

1. R. J. Bayardo, Jr., W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. InfoSleuth: Agent-based semantic integration of information in open and dynamic environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26,2, pages 195–206, New York, 13–15 1997. ACM Press.
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, I. Constantinescu, and B. Faltings. Directory support for large-scale, automated service composition. In *Software Composition*, volume 3628 of *Lecture Notes in Computer Science*, pages 57–66, Edinburgh, Scotland, Apr. 2005. Springer.
4. A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300, 1997.
5. BPEL4WS. Business process execution language for web services version 1.1, http://www.ibm.com/developerworks/library/ws-bpel/.
6. 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.
7. A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.
8. I. Constantinescu, W. Binder, and B. Faltings. An extensible directory enabling efficient semantic web service integration. In *3rd International Semantic Web Conference (ISWC 2004)*, pages 605–619, Hiroshima, Japan, Nov. 2004.
9. I. Constantinescu, W. Binder, and B. Faltings. Flexible and efficient matchmaking and ranking in service directories. In *2005 IEEE International Conference on Web Services (ICWS-2005)*, pages 5–12, Florida, July 2005.
10. I. Constantinescu and B. Faltings. Efficient matchmaking and directory services. In *The 2003 IEEE/WIC International Conference on Web Intelligence*, pages 75–81, 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)*, pages 506–513, San Diego, CA, USA, July 2004.
13. A. Doan and A. Y. Halevy. Efficiently ordering query plans for data integration. In *ICDE*, 2002.
14. D. Fensel, W. Wahlster, and H. Lieberman, editors. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*. MIT Press, Cambridge, MA, USA, 2002.
15. R. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *IJCAI*, pages 608–620, 1971.
16. F. Giunchiglia and P. Traverso. Planning as model checking. In *European Conference on Planning*, pages 1–20, 1999.
17. International Conference on Planning and Scheduling. International Planning Competition, http://ipc.icaps-conference.org/.
18. U. Kuter and D. S. Nau. Forward-chaining planning in nondeterministic domains. In *AAAI*, pages 513–518, 2004.
19. 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.
20. 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.
21. 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. 2002. Morgan Kaufmann Publishers.
22. OWL-S. DAML Services, `http://www.daml.org/services/owl-s/`.
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. E. P. D. Pednault. ADL: Exploring the middle ground between strips and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332, Morgan Kaufmann Publishers, 1989.
25. S. R. Ponnekanti and A. Fox. Sword: A developer toolkit for web service composition. In *11th World Wide Web Conference (Web Engineering Track)*, 2002.
26. 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.
27. P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *International Semantic Web Conference*, volume 3298 of *Lecture Notes in Computer Science*, pages 380–394. Springer, 2004.
28. W3C. OWL Web Ontology Language 1.0 Reference, http://www.w3.org/tr/owl-ref/.
29. W3C. RDF Primer, http://www.w3.org/tr/rdf-primer/.
30. W3C. XML Schema Part 2: Datatypes, http://www.w3.org/tr/xmlschema-2/.
31. WSMO. Web Service Modeling Ontology, http://www.wsmo.org/.
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.