Optimizing the Tradeoff between Discovery, Composition, and Execution Cost in Service Composition

Immanuel Trummer and Boi Faltings
Artificial Intelligence Laboratory
Ecole Polytechnique Fédérale de Lausanne
Switzerland
{firstname}.{lastname}@epfl.ch

***Abstract*—Quality-aware service composition starts from an abstract workflow. The tasks of the workflow are associated with functional types for which concrete services can be retrieved from a registry. Abstract tasks have to be mapped to concrete services before the workflow is executed. The goal is to maximize the workflow quality by choosing the right combination of services.**

Spending more time in discovery and composition will increase the quality of the resulting workflow. Restricted resources motivate however the question about the optimal tradeoff between composition effort and solution quality. In this paper, we aggregate the three phases discovery, composition, and execution into a common cost metric. We motivate why this cost metric may dynamically change depending on the system state and the properties of the workflow at hand. We present and analyze an iterative algorithm that automatically balances the effort spent in different phases. We are able to prove a near-optimal number of iterations. Additionally, we provide extensive experimental evaluations showing that our algorithm significantly outperforms static approaches in dynamic scenarios.

I. Introduction

Web Services are self-contained, self-describing, modular applications which can be discovered and invoked over the Internet. They can be composed in workflows using languages like BPEL4WS [1]. The process of generating automatically a workflow that achieves a given goal is called *service composition* [2]. Considering not only the functionality but also the quality of the composition leads to quality-aware service composition. In this context, it is often assumed that a workflow template is already given whose abstract tasks have to be mapped to concrete service instances in an optimal fashion (e.g. [3]). Answering a user request for composition and execution involves three phases: discovering candidate services from a registry, composing services in an optimal way, and executing the resulting workflow. Literature on service composition mostly focuses on one or two of these phases. Zeng et al. [3] focus on reducing execution time and cost, Constantinescu et al. [4] on reducing the number of directory accesses. Berbner et al. [5] aim at restricting the composition effort while still providing good quality results in most cases.

In this paper, we focus on the efficiency of a system that performs all three phases. We believe that achieving the best overall efficiency requires to choose a good tradeoff between effort spent in discovery, composition, and execution. We will show that the best tradeoff may change dynamically and that an adaptive algorithm is therefore required. The original scientific contributions of this paper are *(i)* an analysis of the tradeoff between service discovery, composition, and execution, *(ii)* an iterative algorithm that dynamically tunes the effort spent in the different phases, and *(iii)* an extensive experimental evaluation of our algorithm in comparison with static approaches. The remainder of the paper is organized as follows. In Sect. II we describe a motivating scenario where the relative importance of the different time components may change dynamically. In Sect. III we describe related work in service composition. Sect. IV introduces our formal model for the problem of service composition and a corresponding formulation as integer linear programming problem. We describe our test suite in Sect. V and analyze several test runs. In Sect. VI we describe and analyze the iterative algorithms in detail. We evaluate our algorithm against static methods in Sect. VII, discuss the results in Sect. VIII, and conclude with Sect. IX.

II. Motivating Scenario

In this paper we focus on systems that provide workflow composition and execution services to the users. Such a system might be used internally within an enterprise or belong to a provider who offers such services to the public. Service composition algorithms often can be tuned via different parameters that have an influence on the resources needed for running the composition algorithm and on the quality of the resulting workflow at the same time. Additionally, the effort for downloading service descriptions from a directory has to be taken into account. With limited resources, the question arises how much effort to spend for composing one specific workflow. We formalize the tradeoff between effort spent in composition and quality of the resulting workflow by integrating the three phases discovery, composition, and execution into a common cost metric. In this section we will introduce the cost metric and show why this metric may change dynamically depending on the context. This motivates the need for an adaptive composition algorithm. Note that we only integrate the execution time of the resulting workflow in

978-0-7695-4463-2/11 $26.00 © 2011 IEEE
DOI 10.1109/ICWS.2011.108

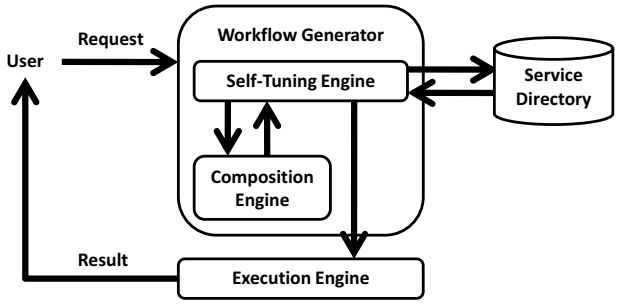

Fig. 1. High-Level Architecture

our cost metric while our approach could be generalized to take into account other quality properties as well.

Fig. 1 shows the architecture on which we base our analysis. Requests consist of a workflow model whose abstract tasks have to be mapped to concrete services prior to execution. Additionally, users may specify a desired number of executions (for instance once for every item of a dataset) as well as certain global quality requirements on the concrete workflow. User requests are processed in three stages. First, the abstract workflow model is used to issue requests to the *Service Directory*, retrieving a set of candidate services for every abstract task. Second, the *Composition Engine* selects one concrete service for every task such that the global quality requirements are respected while the execution time is minimized. Third, the concrete workflow is sent from the *Workflow Generator* to the *Execution Engine*—expressed in a language like BPEL4WS [1]—where it is executed. The result is sent back to the user.

For a given query, we designate by t_r the time it takes to retrieve service descriptions from the registry, by t_c the time it takes to select concrete services, and by t_e the time it takes to execute the workflow once. Approaches to quality-aware service composition like [3] aim at optimizing the concrete workflow by downloading all candidate services and solving the composition problem optimally. However, the registry is a shared resource that is accessed remotely and may contain a large number of services. Also, the problem of quality-aware service composition (as formulated for example in [3], [6]) is NP-hard and solving it optimally may be very expensive. Therefore, t_c and t_r may be non-negligible in comparison to t_e and focusing on t_e alone might not be practical.

Other approaches like [5], [6] propose the use of computationally less expensive methods for the composition which may yield a suboptimal solution while reducing the composition time. However, there are cases where the workflow execution will take a long time (scientific workflows for sophisticated simulations for instance) or where it is executed many times. In these cases, increasing t_c and t_r in order to minimize t_e would be beneficial. This is why we advocate the use of an additional component, the *Self-Tuning Engine*, that dynamically adapts the tradeoff between t_r, t_c, and t_e for a given request in order to minimize the total cost in (1). $\vec{x}$ designates a vector of parameters that the engine can use to tune the amount of downloaded services and the effort of composition planning (which will indirectly influence t_e as well). We assume that the time for calculating the optimal tradeoff can be neglected. In the following, we will discuss two scenarios that motivate different choices of the weights w_r, w_c, and w_e.

$$c(\vec{x}) = w_r t_r(\vec{x}) + w_c t_c(\vec{x}) + w_e t_e(\vec{x}) \quad (1)$$

$$\frac{n_q}{\max\{T_r, T_c, T_e\}} \quad (2)$$

A. Minimize Response Time for Single Query

In this scenario, the system processes only one query at the same time. Discovery must be executed prior to composition, and composition is required before the workflow is executed. Hence, the response time perceived by the user corresponds to the sum of the corresponding three time components. We set $w_r = w_c = 1$. t_e is the time for one workflow execution. The total time for execution depends on the number of desired workflow executions—n_e—as well as on certain system properties. If the execution engine supports for example up to 50 concurrent workflow executions, we choose $w_e = \lceil \frac{n_e}{50} \rceil$. On the other hand, if the execution can be entirely parallelized, we set $w_e = 1$.

B. Maximize System Throughput

In this scenario, the system steadily receives new queries from multiple users. Incoming requests are queued until they can be processed. In order to avoid a growing queue, we want to maximize the throughput of the system. For optimally exploiting the available resources, the three stages of query processing can be pipelined. Hence, the system can for example already download service descriptions for the next user request, while the current user request is still being processed in the composition engine.

We assume that the system is always saturated with requests (the first processing stage, the service download, is never idle). We consider a time interval which is sufficiently long for neglecting the time for filling up the pipeline and termination. We assume that we have fixed weights w_r and w_c and that w_e only depends on the number of desired workflow executions. We assume that n_q queries have been processed during the considered time interval. The throughput of the system can be expressed by (2) where T_r, T_c, and T_e express the total time for registry access, composition, and execution.

In order to maximize the throughput, we have to minimize the expression $\max\{T_r, T_c, T_e\}$. It is possible that one of the time components is significantly larger than the others. Choosing a higher weight (w_r, w_c, w_e) for the corresponding stage will motivate the self-tuning engine to spend less time in the corresponding stage. This will tend to improve the overall throughput as well. Which processing stage becomes the bottleneck, depends on the nature of the issued queries as well as on the properties of the system (connection to service directory, hardware on which the composition engine is

running). These may also dynamically change, if for example the nature of the queries changes or the registry—which is a shared resource—is slowed down by too many queries from different composition engines. Also, the system may switch between throughput maximization and minimization of response time depending on the frequency of the incoming user queries. Altogether, this motivates the use of an algorithm that dynamically adapts the tradeoff between registry access, composition, and execution time to the query at hand and to the overall system state.

III. Related Work

Zeng et al. [3] as well as Aggarwall et al. [10] propose the use of integer linear programming (ILP). This method always finds the best solution but the solved problems are NP-hard. Therefore, the scalability is limited. Different publications addressed the same or very similar problems by computationally less expensive optimization algorithms, while accepting lower quality of the solutions. Canforra et al. [6] use genetic algorithms. Berbner et al. [5] propose heuristics for the same problem and Yu and Lin [11] compare the tradeoff between optimization time and solution quality for different heuristic and non-heuristic algorithms.

Gu et al. [12] present a distributed peer-to-peer approach to service composition. In their approach, functional service graphs can have several permutations that may allow solutions with different quality. One feature of their system is that the number of considered graph permutations can be tuned. A higher number of permutations increases the required running time for the composition algorithm but may also increase the quality of the result. Klein et al. [13] present several heuristic algorithms for quality-aware service composition. Their algorithms can also be tuned via different parameters controlling how thoroughly the search space is explored. The two latter approaches present algorithms for quality-aware service composition that can be tuned, realizing different tradeoffs between composition effort and solution quality. The iterative schema that we will present in the following could be combined with these and other service composition algorithms.

Many approaches to service composition do not consider the effort of discovery. This does not seem realistic in the case of large-scale registries, as was already noted by Constantinescu et al. [4]. They present an approach to functional service composition that performs interleaved composition and service discovery in order to minimize the number of directory accesses. The approach that we present in this paper would generally benefit from mechanisms that filter the services in the registry in order to reduce the number of considered service candidates. Alrifai et al. [14] introduce the notion of skyline services. These are services for which no other service exists which is as least as good in every quality dimension and strictly better in at least one. Alrifai et al. prove that the optimal composition can only contain skyline services so other services do not need to be considered.

Our approach is also related to work in database systems on the tradeoff between query optimization and execution. The general idea of adapting the planning effort to the execution complexity was patented [15]. Shekhar et al. [16] apply semantic transformations to a query until one of several stopping criteria is satisfied. The stopping criteria are based on the relation between optimization and execution time, one of them integrates the principle of diminishing returns as we do. However, they do not consider issues specific to web service composition like the additional time for discovery and a growing search space (instead of several separate ones).

IV. Service Composition Model

In this section, we present a formal model for quality-aware service composition and a corresponding formulation as integer linear programming problem (ILP) [17]. Our contribution is an algorithm that balances the effort of discovery, composition, and execution. Our contribution is not in the way we address the service composition problem in itself. Therefore, we adopt a formal model and a formulation as ILP which is close to existing work (e.g. [3]) and cover it quickly.

The input to the composition problem is an abstract workflow $\mathcal{W}_{abs}$. We assume that it consists of a sequence of tasks $t_1, ..., t_k$ that have to be executed one after the other. For every task, a set of concrete services is available that can all fulfill the required functionality. We denote by S_j the set of concrete services that could fulfill task t_j ($1 \leq j \leq k$). The concrete services $s_{ij} \in S_j$ expose different quality criteria (for example availability or the quality of the provided output data). In this paper, we will consider m abstract quality criteria $q_1, ..., q_m$ with percentage value domains. By $q_l(s_{ij}) \in \{1\%, ..., 100\%\}$ we denote the concrete quality properties of a given service s_{ij}. In our model, a higher value of q_l always corresponds to the higher quality (corresponding scaling can be applied if this is not the case). Additionally we assume that every service is associated with a response time (in milliseconds) $t(s_{ij})$ with positive integer value domain.

Mapping the tasks of the abstract workflow to concrete services yields a concrete workflow $\mathcal{W}_{con} = (\mathcal{W}_{abs}, M)$ where $M = (t_1, s_{i_1 1}), ..., (t_k, s_{i_k k})$ describes the mappings from tasks to concrete services. We also write $M(t_j)$ for designating the service $s_{ij} \in S_j$ that task t_j was assigned to. The quality properties $q_l(\mathcal{W}_{con})$ and response time $t(\mathcal{W}_{con})$ of a concrete workflow $\mathcal{W}_{con}$ can be calculated in function of the properties of the services used. Dumas et al. [18] identify three classes of QoS attributes, depending on the way they are aggregated: Additive, multiplicative, and attributes whose values are determined by the services on the critical path (e.g. summation over the attributes of the critical services). We assume that every quality property of the concrete workflow is calculated as the sum over the corresponding quality properties of the used services. Multiplicative aggregation can be transformed into a sum by using the logarithm function as detailed by Zeng et al. [3]. We also assume that the running time of the whole workflow (for one invocation) can be calculated as sum over the cost of all single services. In addition to global quality requirements, local compatibility constraints between service candidates could exist that do not allow certain service

combinations. We integrated constraints on the compatibility of input and output format of services as representatives for this type of local constraint. We denote by $f_{in}(s_{ij})$ the input format and by $f_{out}(s_{ij})$ the output format of a service s_{ij}.

The goal of the composition process is to find one concrete workflow $\mathcal{W}_{con}^{opt}$ that has the following three properties. First, it respects certain minimum quality requirements r_l with $1 \leq l \leq m$ set by the user who issued the request: $q_l(\mathcal{W}_{con}^{opt}) \geq r_l$. Second, it uses only services whose input and output formats are compatible. This can be expressed as $\forall 2 \leq j \leq k : f_{out}(M(t_{j-1})) = f_{in}(M(t_j))$. Third, its running time is minimal among all concrete workflows that satisfy conditions 1 and 2.

The composition problem as formulated above corresponds to a Knapsack problem [19]. Hence it is NP-hard and the use of ILP is justified. In the following, we explain how we transform our service composition problem into an ILP. An ILP consists of a set of variables with associated value domains, a set of linear constraints, and an objective function (also linear) that has to be minimized or maximized. We introduce variables y_{ij} with binary domain that represent whether task t_j is mapped on service s_{ij} in the optimal composition ($y_{ij} = 1$ in this case, $y_{ij} = 0$ otherwise). We introduce the following constraints. First, we cover the fact that each task has to be mapped to exactly one concrete service—see (3). Second, we integrate the quality constraints imposed by the user—see (4). Third, we have to take into account the input and output formats of the concrete services and make sure that the output format of every service matches the input format from the following service—see (5). Finally, we integrate the objective which is to minimize the total execution time of the workflow, corresponding to the sum over all service invocations—see (6).

$$\forall j \in \{1, ..., k\} : \sum\nolimits_{i=1..|S_j|} (y_{ij}) = 1 \tag{3}$$

$$\forall l \in \{1, ..., m\} : \sum\nolimits_{j=1..k} (q_l(M(t_j))) \geq r_l \tag{4}$$

$$\forall j \in \{2, ..., k\} : \sum\nolimits_{i=1..|S_{j-1}|} (y_{i,j-1} f_{out}(s_{i,j-1})) = \sum\nolimits_{i=1..|S_j|} (y_{ij} f_{in}(s_{ij})) \tag{5}$$

$$minimize \sum\nolimits_{j=1..k} \sum\nolimits_{i=1..|S_j|} (y_{ij} t(s_{ij})) \tag{6}$$

V. The Test Suite

We implemented a test suite in order to experimentally evaluate different approaches for optimizing the tradeoff between discovery, composition, and execution time. Our test suite generates sequential workflows and sets of corresponding, concrete services for every task. The concrete services for every task are retrieved in a fixed (but not specific) order from the registry. The quality properties and input-/output formats of the services are chosen randomly. For calculating the running time, we integrated the following two factors in order to make our test suite more realistic. First, time and quality are correlated. This means that services of high quality are likely to be relatively expensive. Then, the cost distribution of services depends on the task they accomplish. Invoking simple services looking up zip codes for city names is probably less expensive than invoking services executing complex simulations.

The experiments were executed on a machine with 2.53 GHz Intel Core 2 Duo processor with 2.5 GB RAM running Windows 7. The test case generator was implemented in Java version 1.6. Generated composition problems are transformed into ILPs as explained in the previous section. We used IBM ILOG CPLEX V12.1 as ILP solver. We set the global default thread count (parameter $Threads$) to 1 and solved optimally ($RelObjDif = 0.0$). For all other CPLEX parameters, the default values were used.

A. Detailed Description of Test Case Generation

In the following, we will present the set of parameters of our test suite together with admissible value domains. During the experimental evaluation, these parameters are either set to specific values or chosen randomly with uniform distribution in the admissible value domain. The quality properties q_l of the services in the registry are generated randomly in the range between 0% and 100%. We consider $m = 10$ quality criteria. The possible values for the response time of a service depend on its functional category. The minimum response time for services in a given category is calculated as $t_{min} = 50 \cdot x$ where x is specific to the category and between 1 and 10. The maximum response time for the services is calculated similarly as $t_{max} = 500 \cdot x$. The probability distribution for the response time of a specific service depends on its average quality (arithmetic average over all quality attributes that have been chosen before). In a first step, the response time is chosen as $t = q_{average}(t_{max} - t_{min}) + t_{min}$. Then, a percentage between 0% and 75% is randomly chosen and this percentage is either added or subtracted from t (without leaving the admissible response time interval for the category). The input and output format of services is chosen randomly out of 5 possible formats. When generating abstract workflows, the tasks are randomly associated with one service category.

For the benchmarks, we measure service response times and the time for composition in milliseconds. We assume that the time for registry accesses is proportional to the number of downloaded services and that downloading 1 service takes 10 milliseconds (e.g. for XML description of interface and properties with size 20 KB and a 16000 kbit/s connection).

B. Experimental Evaluation and Analysis

In the following, we will present a first experimental evaluation of our benchmark. We studied the development of execution time, composition time, and composition success rate for different numbers of tasks, different numbers of services per task, and different quality requirements. In particular, we want to show the dependency between composition and execution time. We report the arithmetic average of 50 generated test cases for every parameter set (combination of specific quality requirements, number of services per task, and number of

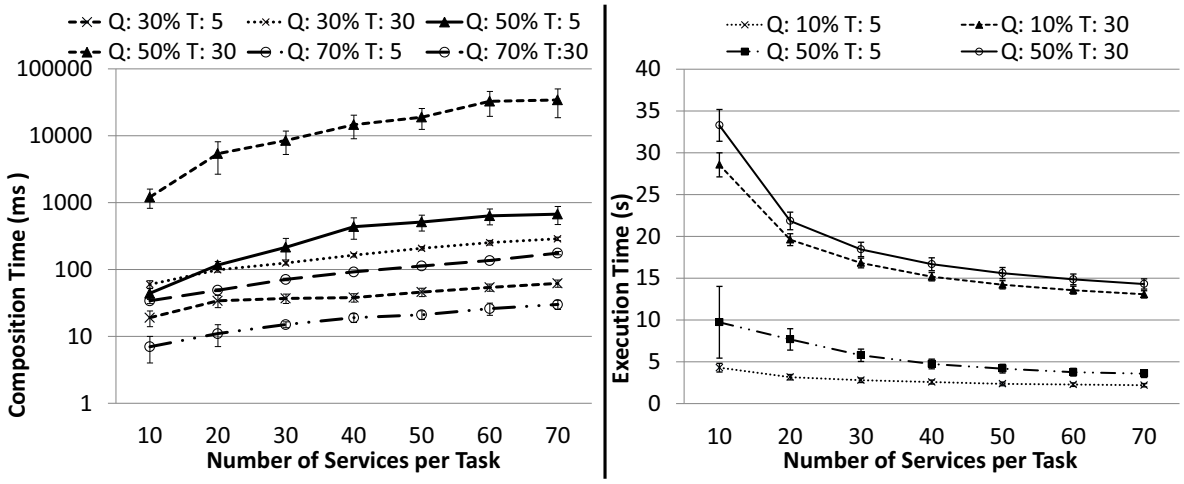

Fig. 2. Dependency between Composition and Execution Time

tasks). Additionally we report the 95 % confidence intervals. We abbreviate the quality requirements on all quality attributes of the workflow by *Q*, the number of tasks by *T*.

Fig. 2 shows on the right side the development of the execution time of the least expensive workflow that can be found within the current search space. We do not count cases in which no solution was found. The size of the search space depends on the number of services that are downloaded per task. We observe the following tendencies. *(i)* We claimed in the beginning of this section that our benchmark respects a correlation between the quality of the single services and their invocation time. This is mirrored by the fact that the execution time for the optimal concrete workflow is higher if the quality requirements are higher. *(ii)* A higher number of tasks leads to higher execution time since more service invocations have to be performed. *(iii)* The execution time is reduced by increasing the number of services per task that are considered. However, the reduction that can be achieved by adding a constant number of new services (10) decreases (*diminishing returns*). This seems logical since the set of services which is already included becomes more and more representative and large improvements become less and less likely. Also the ratio $(N+10)^k/N^k$ (where N is the current number of services per task and k the number of tasks) between the sizes of two consecutive search spaces decreases with increasing N. We will use this fact later in the design of an iterative solution strategy. Note that we describe a statistical tendency which may not be satisfied in every single case.

Fig. 2 shows on the left side the development of the composition time in milliseconds (logarithmic scale) when using complete search. We observe the following tendencies. *(i)* The planning time increases with growing number of services per task which corresponds to a larger search space. *(ii)* The time increases with growing number of tasks since this corresponds to a larger search space, too. *(iii)* The relation between quality requirements and composition time is more ambiguous. For relatively low (30 %) or relatively high (70 %) quality requirements, the composition time is in the order of below 1 second. For quality requirements of 50 %, the planning time reaches 28.5 seconds. For high quality requirements, only few test cases were solvable and infeasibility seems to be detected quickly. For low quality requirements, solutions are distributed densely and good solutions are found quickly. These can be used to prune the search space which speeds up the search.

TABLE I
NUMBER OF SOLVED TEST CASES OUT OF 50

Quality	SPT	5 Tasks	15 Tasks	30 Tasks
30%	10	50	50	50
	20	50	50	50
50%	10	4	44	50
	20	40	50	50
70%	10	0	0	0
	20	0	0	0

It is not always possible to solve a test case with the first chunk of services. Table I shows the number of solved test cases (out of 50 generated test cases each). Clearly, the number depends on the global quality requirements—for higher requirements, less instances can be solved.

VI. TRADEOFF OPTIMIZATION ALGORITHM

We identified three phases in the processing of user queries: service discovery, composition, and execution. Composition requires prior discovery and execution requires prior composition. However, the effort spent in discovery and composition should be adapted to the cost of workflow execution. This cost is difficult to estimate without performing some kind of prior planning. In order to comply with these constraints, we adopt an iterative algorithm. The structure is represented in Fig. 3. The algorithm downloads a few services, performs a limited amount of composition and uses the results to decide whether further discovery and composition steps seem appropriate.

This algorithm would be executed by the self-tuning engine (referring to the architecture from Fig. 1). In the following,

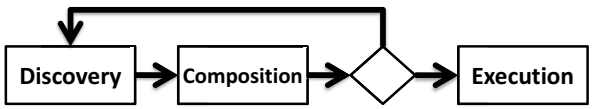

Fig. 3. Tradeoff Optimization Algorithm Overview

we will present the algorithm in more detail. The algorithm uses the principle of diminishing returns in oder to derive an upper bound on the improvements that can be realized by additional iterations. We are able to prove that the number of iterations performed by our algorithm is near optimal under these assumptions.

A. Description of Dynamic Algorithm

We call our algorithm dynamic because it dynamically adapts the effort spent in composition in order to optimize the overall performance. We adopt a representation that resembles object-oriented programming for describing the algorithm. The algorithm receives as input an object D representing the service directory, C corresponds to the composition engine, n_0 is the number of services downloaded per task in every iteration. $\mathcal{W}_{abs}$ is the abstract workflow, r_i the global quality requirements, and w_r, w_c, w_e the current weights. The parameter θ indicates how thoroughly the current search space is explored. It fixes the tradeoff between discovery and composition time. The semantic of the parameter depends on the method which is used to perform the composition. For our benchmarks we use integer linear programming and θ corresponds to the maximum acceptable gap between the utility values of returned and optimal solution (percentage).

In the initialization, we set $\mathcal{W}_{con}$ (symbolizing the concrete workflow) to ϵ indicating that no solution was found so far. S is the set of downloaded service descriptions and is initially empty. In every iteration, we first request n_0 new services for every task in $\mathcal{W}_{abs}$ from the registry (we assume that $\mathcal{W}_{abs}$ is associated with an ID that allows the registry to decide which services to return next). We call the composition engine with the augmented set of services, the abstract workflow, and the quality requirements. Additionally, we provide $\mathcal{W}_{con}$—the solution found in the last iteration—as starting point in order to speed up the search. In our implementation, we therefore used the CPLEX function $\mathrm{addMIPStart}$ with the y_{ij} values of the best solution found in the last iteration (if any).

The solve method returns a concrete workflow $\mathcal{W}_{con}$ with minimum execution time net (new execution time) or $\mathcal{W} = \epsilon$ and $net = \infty$ if no solution has been found. We calculate the cost for the directory access and for composition in the current iteration (using the $\mathrm{getTime}$ function of the corresponding object that returns the time needed for the last operation). We assume that this corresponds to a lower bound for the time needed in the next iteration. This seems justified since we are downloading a constant number of services per task and we execute a complete optimization algorithm within a steadily growing search space. In order to decide whether the next iteration could be beneficial, we also need an upper bound on the reduction in execution cost that could be achieved. Since we assume diminishing returns for always adding the same number of services, we take the reduction that was achieved in this iteration as upper bound for the next one. Additionally, (since execution cost is always positive) we eventually tighten this upper bound by the current execution cost. We do not perform the next iteration if *(i)* either no new services can be retrieved from the registry ($\mathrm{D.empty}() = \mathbf{true}$), or *(ii)* a solution was found ($\mathcal{W}_{con} = \epsilon$) and the minimum cost of the next iteration (ic) cannot outweigh the maximum possible improvement in execution cost (ecr). Finally, we return the solution together with the execution time.

Algorithm 1 Tradeoff Optimization Algorithm

Require: $D, C, n_0, \mathcal{W}_{abs}, r_i, w_r, w_c, w_e, \theta$
1: {Initialization}
2: $W_{con} \leftarrow \epsilon$
3: $S \leftarrow \emptyset$
4: {Until registry empty or solution found and effort of further improvements would probably outweigh benefits}
5: **repeat**
6: {Retrieve new services from directory}
7: $S \leftarrow S \cup \mathrm{D.downloadNext}(\mathcal{W}_{abs}, n_0)$
8: {Solve new optimization problem}
9: $(\mathcal{W}_{con}, net) \leftarrow$ $\mathrm{C.solve}(\mathcal{W}_{abs}, r_i, \mathcal{W}_{con}, \theta)$
10: {Cost of this iteration - lower bound for next}
11: $ic \leftarrow w_r \cdot \mathrm{D.getTime}() + w_c \cdot \mathrm{C.getTime}()$
12: {Upper bound on next execution cost reduction}
13: $ecr \leftarrow w_e \cdot \min(oet - net, net)$
14: {Save execution time for next iteration}
15: $oet \leftarrow net$
16: **until** $\mathrm{D.empty}()$ **or** ($W_{con} \neq \epsilon$ **and** $ecr < ic$)
17: **return** $(\mathcal{W}_{con}, net)$

B. Analysis of Dynamic Algorithm

We prove by the following theorem that our algorithm always performs a number of iterations which is near to one of the optimal ones (there may be severals, each realizing a different tradeoff between the different cost components). We base our proof on the assumption of diminishing returns. This means that the reduction in execution cost achieved by iteration i_2 cannot be higher than the one achieved in i_1 for any $i_1 < i_2$. Additionally, we assume that the cost for discovery and composition during one iteration are non-decreasing as we justified before.

Theorem 1. *Let i the total number of iterations performed by our algorithm and I_{opt} the set of iteration numbers that yield minimum total cost (if no solution was found this corresponds to infinite execution costs). We have $\exists i_{opt} \in I_{opt} : i \geq i_{opt}$ and $\exists i_{opt} \in I_{opt} : i \leq i_{opt} + 1$ in this case. So our algorithm executes at most one iteration more than optimal (and never less iterations).*

Proof: Assume that $\forall i_{opt} \in I_{opt} : i < i_{opt}$. If our algo-

rithm did not perform more than i iterations, this means that either *(i)* no new services can be retrieved, or *(ii)* that we have $ecr < ic$. In the first case, further iterations cannot improve the total cost. Hence our algorithm performs an optimal number of iterations. Now we examine the second case. We assume that ecr is an upper bound for the reduction of the execution cost that can be achieved in the next iteration $i+1$. We also assume that ic is a lower bound on the registry access and composition cost for iteration $i+1$. Hence we have the following relation between the total cost c_i before executing the $(i+1)$th iteration (which our algorithm does not) and c_{i+1} after executing the $(i+1)$th iteration: $c_{i+1} - c_i \geq ic - ecr > 0$. We also assume that ic does not decrease and that ecr does not increase from one iteration to the next. Hence we have $c_i < c_{i+1} \leq c_{i+1}...$ and therefore $\forall j > i : j \notin I_{opt}$ which is a contradiction.

Now assume that $\forall i_{opt} \in I_{opt} : i > i_{opt} + 1$, we set $j = \max(I_{opt})$. Since $(j+1) \notin I_{opt}$ we have $c_{j+1} > c_j$. This cannot be true if no solution was found after the jth iteration. Therefore, we have $ic > |w_e \cdot \delta t_e|$ after the $(j+1)$th iteration (where δt_e designates the improvement on execution time by the $(j+1)$th iteration). We also have $ecr \leq |w_e \cdot \delta t_e|$. This implies that $ic - ecr \geq c_{j+1} - c_j > 0$ but in this case the algorithm terminates. So we have $i = j + 1$ which is a contradiction. ■

VII. Experimental Evaluation

In this section, we compare our dynamic algorithm with several static methods that always download the same number of services and search for the optimal solution. These static methods correspond to different levels of effort which is spent in discovery and composition. Methods that only download few services correspond to heuristics. They reduce cost for discovery and composition but probably yield a suboptimal solution. Methods that download all services (70 services per task in our benchmarks) and perform ILP optimization without time restrictions will yield the optimal solution. On the other side they maximize the time for discovery and composition.

We simulated different scenarios corresponding to different settings for the weights w_r, w_c, and w_e. Table II summarizes the settings for the seven scenarios. We additionally choose the number of tasks of the abstract workflow randomly between 5 and 30 for every test case and randomly associate tasks with categories. The quality requirements are chosen randomly between 1 % and 50 %. In Fig. 4 we report the average aggregated cost values for 100 test cases. The methods *Dynamic(0%)* and *Dynamic(25%)* correspond to our iterative algorithm with values 0% and 25% for parameter θ. We do not report absolute values but scale all values according to the average cost of our algorithm *Dynamic(0%)* which corresponds to 100 %. We tested the case of 10, 20, 30, 40, 50, 60, and 70 services per task but do not report all of them due to space restrictions. We verified that the general tendencies remain the same. The experiments were executed on the same hardware and software platform as described in Sect. V.

We make the following observations concerning the comparison of different static methods. *(i)* The static algorithms

TABLE II
Weights for Different Scenarios

Scenario	w_r	w_c	w_e
rce	1	1	1
Rce	100	1	1
rCe	1	100	1
rcE	1	1	100
RcE	100	1	100
rCE	1	100	100
RCe	100	100	1

that perform less discovery and composition are better (in comparison to other static algorithms) in scenarios where execution time is relatively unimportant. This concerns in particular the scenarios Rce, rCe, and RCe. However, static methods that download only few services may not always find a solution even if one exists. During our test runs, 98 % of the test cases were solvable with only 10 services per task in average (all of them were solvable by integrating more services). *(ii)* Static approaches that download all services and search for the optimal solution perform well (in comparison to other static approaches) in cases where the execution time is of highest importance. This concerns in particular the scenarios rcE and rCE. For scenario RcE, execution time is important but registry access is expensive as well. The additional registry access cost outweigh the gains in execution cost here.

We make the following observations concerning the performance of the dynamic algorithms. *(i)* Comparing the static approaches with *Dynamic(0%)*, we see that every static approach has one or several scenarios where it performs significantly worse than the adaptive algorithm. Also note that our algorithms always found a solution for the test cases while the method that only downloaded 10 services per task did not. *(ii)* We verified that there is at least one scenario for every static method where the aggregated cost sum up to at least 188% of the cost of *Dynamic(0%)*. In many cases the gap is significantly larger (up to over 700% for scenario RCe and 70 services per task for example). On the other hand, our algorithm is most significantly outperformed in scenario rCe by the static method that downloads 10 services per task. This method achieves 72% of the cost of our algorithm. *(iii)* Comparing *Dynamic(0%)* and *Dynamic(25%)*, we find that *Dynamic(25%)* is better in scenario rCE as can be expected since it reduces the effort of composition. In scenarios where execution time is important, *Dynamic(0%)* is preferable.

VIII. Discussion

In Sect. II we named various reasons for a change of the weights in the cost metric. These include changes in the system state (connection to the registry, frequency of requests) as well as properties of the incoming requests (how many workflow executions are planned, order of magnitude of execution time). In the experimental evaluation we have shown that our algorithm is able to dynamically adapt to changing priorities. Static approaches only perform well in specific situations. Our iterative schema is based on the following assumptions: *i)* Non-Decreasing effort of optimization when increasing the

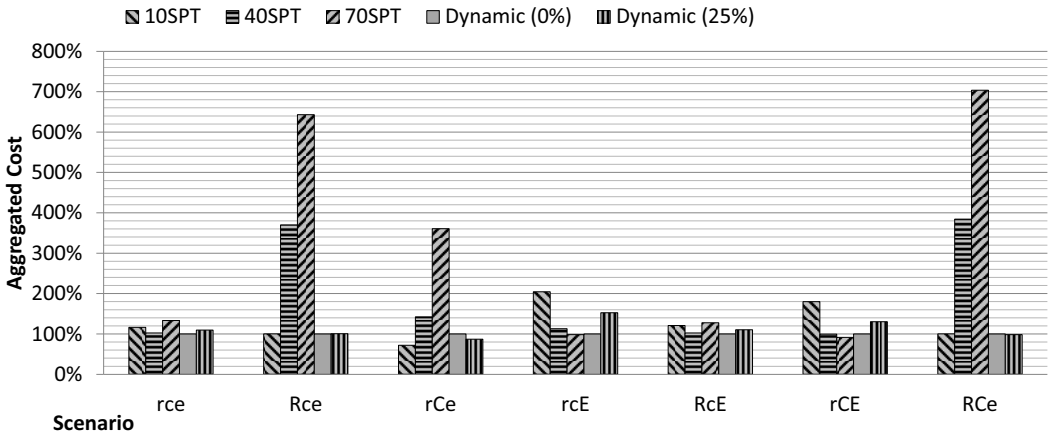

Fig. 4. Comparison of Static and Dynamic Methods for Quality-Aware Service Composition

size of the search space, *(ii)* increasing quality of the resulting workflow when adding new services, *(iii)* diminishing returns when adding more and more services. These are generic principles and not specific to the workflow and service model that we use in this paper. Additionally, the iterative schema that we propose could be extended to other underlying composition algorithms. Integer linear programming is convenient because it allows to reuse solutions from previous iterations and to tune the effort of search space exploration. However, other methods offering similar possibilities could be used as well.

IX. Conclusion

We extended existing literature on service composition by addressing the efficiency of service discovery, service composition, and service execution as a whole. The optimal tradeoff depends on different factors and may change dynamically. We presented an iterative schema that automatically balances the tradeoff between the different phases. We are able to prove a near-optimal number of iterations under reasonable assumptions. Experimental evaluations show, that our algorithm significantly outperforms static approaches.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "SOSOA: Self-Organizing Service-Oriented Architectures" (SNF Sinergia Project No. CRSI22_127386/1). We also thank the four anonymous reviewers for their very useful feedback.

References

[1] "Web services business process execution language version 2.0," *URL: http://docs. oasis-open. org/wsbpel/2.0/OS/wsbpel-v2. 0-OS. html*, 2007.

[2] J. Rao and X. Su, "A survey of automated web service composition methods," *Semantic Web Services and Web Process Composition*, pp. 43–54, 2005.

[3] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "QoS-aware middleware for web services composition," *Software Engineering, IEEE Transactions on*, vol. 30, no. 5, pp. 311–327, 2004.

[4] I. Constantinescu, W. Binder, and B. Faltings, "Service composition with directories," in *Software Composition*. Springer, 2006, pp. 163–177.

[5] R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz, "Heuristics for QoS-aware web service composition," in *Web Services, 2006. ICWS'06. International Conference on*. IEEE, 2006, pp. 72–82.

[6] G. Canfora, M. Di Penta, R. Esposito, and M. Villani, "An approach for QoS-aware service composition based on genetic algorithms," in *Proceedings of the 2005 conference on Genetic and evolutionary computation*. ACM, 2005, pp. 1069–1075.

[7] S. McIlraith and T. Son, "Adapting golog for composition of semantic web services," in *PRINCIPLES OF KNOWLEDGE REPRESENTATION AND REASONING-INTERNATIONAL CONFERENCE-*, 2002, pp. 482–496.

[8] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau, "Automating DAML-S web services composition using SHOP2," *The SemanticWeb-ISWC 2003*, pp. 195–210, 2003.

[9] S. Ponnekanti and A. Fox, "Sword: A developer toolkit for web service composition," in *Proc. of the Eleventh International World Wide Web Conference, Honolulu, HI*, 2002.

[10] R. Aggarwal, K. Verma, J. Miller, and W. Milnor, "Constraint driven web service composition in meteor-s," in *Services Computing, 2004.(SCC 2004). Proceedings. 2004 IEEE International Conference on*. IEEE, 2004, pp. 23–30.

[11] T. Yu and K. Lin, "Service selection algorithms for composing complex services with multiple QoS constraints," *Service-Oriented Computing-ICSOC 2005*, pp. 130–143, 2005.

[12] X. Gu, K. Nahrstedt, and B. Yu, "Spidernet: An integrated peer-to-peer service composition framework," in *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*. IEEE, 2004, pp. 110–119.

[13] A. Klein, F. Ishikawa, and S. Honiden, "Efficient QoS-Aware Service Composition with a Probabilistic Service Selection Policy," *Service-Oriented Computing*, pp. 182–196, 2010.

[14] M. Alrifai, D. Skoutas, and T. Risse, "Selecting skyline services for QoS-based web service composition," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 11–20.

[15] G. Lohman, K. Ono, and J. Palmer, "System for adapting query optimization effort to expected execution time," Apr. 5 1994, uS Patent 5,301,317.

[16] S. Shekar, J. Srivastava, and S. Dutta, "A formal model of trade-off between optimization and execution costs in semantic query optimization," in *Proc. 14th VLDB*, pp. 457–467.

[17] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons Inc, 1998.

[18] M. Dumas, L. García-Bañuelos, A. Polyvyanyy, Y. Yang, and L. Zhang, "Aggregate quality of service computation for composite services," *Service-Oriented Computing*, pp. 213–227, 2010.

[19] H. Keller, U. Pferschy, and D. Pisinger, "Knapsack problems," 2004.

[20] Amazon ec2 spot instances. [Online]. Available: http://aws.amazon.com/ec2/spot-instances/