Efficient QoS-aware Service Composition with a Probabilistic Service Selection Policy

Adrian Klein[1], Fuyuki Ishikawa[2], and Shinichi Honiden[1,2]

[1] The University of Tokyo, Japan
[2] National Institute of Informatics, Tokyo, Japan
{adrian,f-ishikawa,honiden}@nii.ac.jp

Abstract. Service-Oriented Architecture enables the composition of loosely coupled services provided with varying Quality of Service (QoS) levels. Given a composition, finding the set of services that optimizes some QoS attributes under given QoS constraints has been shown to be NP-hard. Until now the problem has been considered only for a single execution, choosing a single service for each workflow element. This contrasts with reality where services often are executed hundreds and thousands of times. Therefore, we modify the problem to consider repeated executions of services in the long-term. We also allow to choose multiple services for the same workflow element according to a probabilistic selection policy. We model this modified problem with Linear Programming, allowing us to solve it optimally in polynomial time. We discuss and evaluate the different applications of our approach, show in which cases it yields the biggest utility gains, and compare it to the original problem.

1 Introduction

1.1 SOA

Service-Oriented Architecture (SOA) consists of a set of design principles which enable defining and composing interoperable services in a loosely coupled way. The value of SOA lies in assuring such compositions are easily and rapidly possible with low costs. Thus, service composition is a key to SOA. Especially, achieving an automatic service composition remains a major challenge [1].

When selecting a service not only functional requirements, but also the non-functional requirements [2], expressed by Quality of Service (QoS) attributes, are very important. Especially, when there are many functionally equivalent services, the QoS becomes the deciding factor. Thus, QoS-awareness is of crucial importance in service composition.

QoS-aware automatic service composition is employed in two different problem settings: In the planning problem the composition itself is computed by taking into consideration available input data and desired output data. While the heuristics used for the planning can be geared towards optimizing QoS [3], finding a functionally sufficient composition is the main focus.

This contrasts the traditional composition problem[1] where the functional part, the composition itself, is already given, e.g. as a business process specified with BPEL [4], requiring some inputs, invoking some tasks, and producing some desired outputs (Fig. 1).

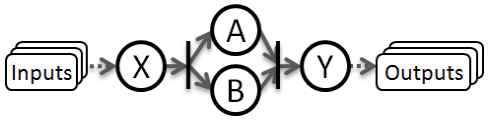


Fig. 1. Sample Composition

The main focus lies in selecting the set of services, with one service per task, to execute the composition optimally with regards to QoS. This means to maximize the overall QoS of the composition according to given preferences while adhering to given QoS constraints [5]. Of course, the output of the planning problem could be used as an input for the composition problem.

1.2 Composition Problem

In this paper we will focus on the latter, the composition problem. For example, if we take a simple workflow consisting of a sequence of two tasks X and Y, and corresponding services S_1, S_2 and S_3, S_4 that can fulfill X and Y, respectively, there are four possibilities to execute this workflow (Fig. 2). Out of those possibilities, the most common approach is to select the optimal set of services regarding QoS preferences and constraints given [5, 6].

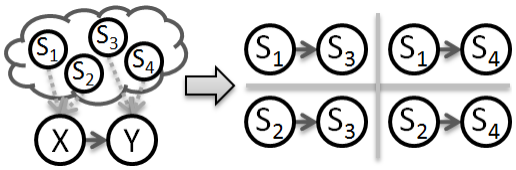


Fig. 2. Sample Composition Problem

Trying all combinations obviously takes exponential time. Modeling it as a Multi-Choice Multidimensional Knapsack problem (MMKP) gives the same result, as MMKP is known to be NP-hard [7]. In order to make use of existing solvers, the problem is usually modeled by Integer (Linear) Programming (IP), which is also NP-hard to solve in the general case. Thus, as of today, the problem cannot be solved efficiently in polynomial time. This is an obstacle to the vision of SOA

[1] We will refer to this problem as the composition problem from now on.

encompassing a future "Enabling a Web of billions of services"[2] where efficient algorithms are crucial in order to enable fast composition queries and adaptation at runtime.

1.3 Contribution

In the present state of the art, approaches for the composition problem implicitly assume that a service is executed only a single time. As a result, one service is statically assigned to each task. We believe that in many situations a service composition will be executed repeatedly. Thus, we propose to optimize the QoS for the long-term, which implies the following:

1. There is no need to choose always the same set of services for each execution.
2. QoS constraints should also be applied for the long-term, e.g. keeping to a monthly budget or assuring a hourly throughput.

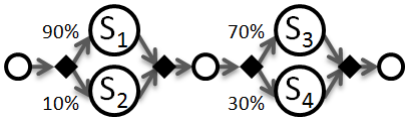


Fig. 3. Sample Probabilistic Composition

For instance, the combinations $\{S_1, S_3\}$ and $\{S_2, S_4\}$ could represent different trade-offs to execute the given workflow: One being fast/expensive and the other cheap/slow. In such a case, for the long-term an optimal solution will most likely contain a probabilistic mix of those two combinations in a certain ratio (Fig. 3) in order to satisfy constraints on budget and throughput while adhering to the QoS preferences given. We propose that such a solution is represented by a set of probabilistic service selection policies. For each task a policy defines the probability that a service is chosen for it at runtime.

As for the computational approach, we propose to use linear programming (LP). We do not restrict our problem to integer solutions, as we represent the decision to choose a certain service with a continuous probability value, not a (binary) integer value. Thus, besides this difference, we can use identical modeling as in IP, and, at the same time, profit from efficient polynomial algorithms to solve the problem. The only caveat is that such probabilistic modeling also changes the nature of the specified QoS constraints, forcing them to apply to the long-term. Later we will discuss the detailed implications of this fact, and show some alternative ways to still guarantee QoS constraints for each execution, if needed.

To summarize, we propose an approach for the service composition problem that is both efficient and QoS-aware, maximizes QoS over the long-term, and

[2] Credo & goal of the SOA4ALL project (part of the European FP7): www.soa4all.eu

results in a set of probabilistic service selection policies. According to this policies the services will be chosen probabilistically at runtime.

The structure of this paper is as follows: Section 2 surveys related work. Section 3 defines our approach. Section 4 explains the possible applications of our approach. Section 5 evaluates the performance and effectiveness of our approach. Finally, Section 6 concludes the paper.

2 Related Work

In this section we survey related work which can be roughly classified into four different categories: work laying the foundation for our approach, work presenting an alternative to our approach, complementary work, and work sharing similar ideas.

The foundation for the QoS-aware composition problem is given in [5]. Many common notions we use are introduced there, and the problem is formalized and solved using IP.

As performance is such an important issue there are many alternative approaches tackling it. One popular theme is to to reduce the search space with heuristics [6, 8, 9]. Another alternative is to use a genetic algorithm to solve the problem [10]. Only optimizing locally is also an option [11], though its conception is slightly opposite to our idea that multiple services can be mixed and (globally) compensate each others QoS. In comparison to our approach, all this approaches are also efficient, but find only approximate solutions to the given optimization problem.

A complementary approach is to compute the skyline of the services involved in the composition beforehand in order to prune services that can never be part of an optimal solution [12]. Obviously, this can be easily integrated with our approach.

There is also work that shares ideas on a conceptual level. In [13] multiple services for the same task are provisioned for adaptation at runtime in the case that one of them fails. For instance, if the first service for a task fails, the second one will be called, and so on. The goal is to improve the failure resilience of a service execution, but not to combine those multiple services by calling them probabilistically at runtime. Regarding considering several executions over the long-term, there is not much related work, but in [14] we already showed an approach that optimizes the service selection for a specific user given his expected usage over the long-term. To the best of our knowledge no related work exists that considers repeated executions of services in the long-term for QoS-aware optimization though.

In conclusion our contribution lies in optimizing the QoS in the long-term efficiently in polynomial time and in selecting between multiple services probabilistically at runtime.

3 Approach

In this section we define our approach formally. The notions, which we will briefly introduce in this section, follow the IP versions of the problem given in [5, 6]. Afterwards, we explain what the consequences of modeling the problem as LP are with regards to the properties that hold for the solutions found.

3.1 Formal Definition

As in [6], the following composition patterns are supported: Seq (sequential execution), AND (parallel execution), XOR (alternative execution with certain probabilities p_i) and Loop (with maximum loop count x).

Fig. 4. Composition patterns: Seq, AND, XOR, Loop

For any given workflow, we have to aggregate the QoS according to the composition patterns. As usual, we unroll any loop to a sequence according to the maximum loop count. Then, the QoS gets aggregated according to the aggregation functions presented in [15]. (Also attributes and their corresponding constraints aggregated by non-linear functions are linearized for the LP formalization, e.g. by taking their logarithm.) Given n aggregated QoS attributes $q_1, \ldots, q_n$, we normalize them like in [5] in order to limit their values to $[0, 1]$ and to be able to simply maximize the weighted sum of their normalized values $\tilde{q_1}, \ldots, \tilde{q_n}$:

$$\sum_{i=1}^{n} w_i \tilde{q_i} \tag{1}$$

Regarding the constraints, we first consider a sample workflow that consists of the four tasks $\{X, Y, A, B\}$ (Fig. 5). A and B are executed in parallel, while the rest is executed sequentially. Thus, each execution of this workflow will execute all four tasks. In addition, there are two paths through the workflow, namely $\{X, A, Y\}$ and $\{X, B, Y\}$. Therefore, we maximize the sum of (1) over all paths, as this has been proven to produce the best results in [6].

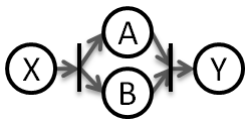


Fig. 5. Sample Workflow

The constraints Q_i^{min}/Q_i^{max} are applied on the aggregated non-normalized QoS of the workflow:

$$q_i \geq Q_i^{min} \wedge q_i \leq Q_i^{max} \tag{2}$$

We follow [6] in that a constraint needs to be applied to the whole workflow, and/or to each path, depending on the particular kind of QoS and the workflow. For example, for response time in conjunction with parallel executions paths, we have to apply the constraint for each path (e.g. $\{X, A, Y\}$ and $\{X, B, Y\}$), while for price we have to apply its constraint to the whole execution (e.g. $\{X, Y, A, B\}$).

Regarding the selection variables, for each task t_i, we introduce as many variables s_{ij} as there are services capable of performing that task. This variables represent the possible choices in our LP problem. In our approach, they represent probabilities and, thus, are not constrained to be integers which is a difference to the notions in [5, 6]. Only their sum must be equal to one for each task, as usual.

3.2 Relaxation

In general, removing the integer restriction on the variables of an IP problem is called relaxing the IP problem. Thus, from now on, we will refer to the former IP problem as the original problem and to our LP problem as the relaxed problem.

One important consequence is that we can solve our relaxed problem with regular linear programming for which efficient algorithms like Simplex or Interior Point exist that solve the problem in polynomial time. This is opposed to the algorithms needed for IP that are exponential in the worst-case as IP is NP-hard. The corresponding caveat is that this changes the nature of our constraints. For instance a solution to the relaxed problem does not necessarily solve the original problem, as seen from a small example workflow consisting of a single task with two services A and B chosen for it according to Table 1. Such a probabilistic combination of A and B will perfectly satisfy a constraint of 100ms on the response time, as the QoS of A and B are multiplied by their probabilities before being compared to the constraint. Thus, individual executions will violate the constraint, even if in the long-term the expected value of the response time fulfills the constraint. We will show some approaches that deal with this issue in the next section where we describe how our approach can be applied. Those enable us to get closer to guaranteeing QoS constraints for each execution, if needed.

Regarding the utility obtained, the solution to our relaxed problem is guaranteed to be at least as good, or better than the solution to the original problem.

Table 1. Example Service Combination

Service	**Probability**	**Response time**
A	50%	150ms
B	50%	50ms

That is because the feasible solutions of the original problem are only a subset of the feasible solutions of the relaxed problem. The absolute difference in utility depends on the formalization of the problem (e.g. scaling of QoS), but the relative difference can become arbitrary large. Intuitively, a constraint could forbid to choose a good solution in the original problem, even if the constraint is only violated slightly. On the contrary, in the relaxed problem we can choose this solution, if we can compensate this small violation in the long-term.

3.3 Result

As for the result of our approach, we get a set of probabilistic service selection policies, as illustrated in Fig. 6. At runtime the services are chosen according to those policies. An approach that also tries to limit the deviation from the policies at any given time should be preferred to a purely random strategy.

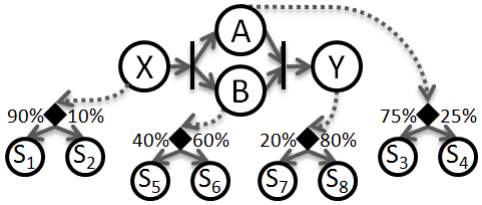


Fig. 6. Probabilistic Service Selection Policy

4 Application

In this section we show four different ways of applying our approach in practice: We apply it to solve the original and the relaxed problem. Also it can be applied to solve a refinement of our relaxed problem, and, as part of a computer-aided decision process, it can be applied, too.

4.1 Original Problem

One interesting application is to use the solution of the relaxed problem to find a solution for the original problem. The advantage of doing this is the performance gain compared to using IP. This raises the questions, if this is possible, and, if such solutions are optimal. To answer these questions, it is important to examine how the solution to the relaxed problem differs from the one for the original problem. As we get a set of probabilistic service selection policies instead of a set of concrete services, a single solution to the relaxed problem contains many potential solutions to the original problem. But, as we know, those solutions only satisfy the QoS constraints in the long-term.

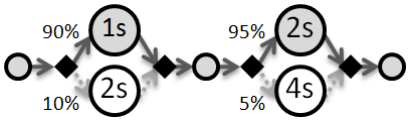


Fig. 7. Choosing a Subset of a Probabilistic Composition

For example, the probabilistic composition in Fig. 7 satisfies a constraint of 3.2 seconds on its response time[3], and to assure this constraint for each execution, we can choose the subset of possible executions colored in grey. Apparently, if we have just a single constraint, a solution that satisfies the QoS constraint for each execution must exist. Given multiple QoS constraints this is no longer true, as no single solution might satisfy all constraints. For example with constraints Q and R, we might have two solutions, one over-satisfying Q, but not satisfying R, and the other one vice versa. Thus, in the long-term their combination may satisfy Q and R, but none of them satisfies both. This means that we may not find such a solution in all cases, especially, if there are many constraints. In general, it is quite difficult to answer how good the quality of such a solution will be compared to the optimal solution. Therefore, we will rely on giving some empirical results later in the evaluation part. One note regarding the scalability: In order to find such a solution, we have to look at all possible solutions induced by the probabilistic policies. If the number of possible solutions becomes too large, it might be necessary to use a Monte-Carlo approach, just probabilistically checking some solutions instead of an exhaustive search in order not to loose the original performance benefit.

4.2 Relaxed Problem

The most obvious application is, of course, to apply the obtained solution to the relaxed problem. We already discussed that this gives us a better utility than solving the original problem. Also the constraints on the QoS are only guaranteed for the long-term. We already mentioned some cases for which this works really well, e.g. for a budget or throughput applied monthly, daily, et cetera. The question is in which cases this is not acceptable. The most general answer is: when a QoS constraint is important for each and every single execution. While it is difficult to imagine a budget, throughput, or availability to be required for every execution, QoS that directly correspond to the experience of the service user might be a concern. One prime example is response time: a user might only tolerate a certain maximum response time when he calls a service. Also, even if a certain deviation is still acceptable, it might affect the impression of how good a service is. In such cases, not only guaranteeing the QoS constraint in the long-term, but additionally assuring only a limited deviation from it seems better than directly taking the solution to the relaxed problem. This takes us to next application of our approach.

[3]The services are annotated with their response times in s(econds) in the figure.

4.3 Refined Relaxed Problem

Applying our approach to solve a refinement of the relaxed problem is the next logical step. The idea is to solve the problem iteratively over and over again, refining it slightly in each iteration to get closer to the desired result, e.g. guaranteeing some QoS constraints for every execution, or at least for a sufficient percentage (e.g. 99.9%) of the executions. If a refinement with a sufficiently good solution can be obtained in a limited (e.g. constant) number of iterations, then we do not even lose the performance benefit either. In general, there are two basic refinements: changing an existing constraint, and adding an additional constraint. Both share the common refinement process shown in Fig. 8.

For example, to guarantee a better response time for each execution we can make the constraint on response time tighter. This will improve QoS in the long-term and also for each execution on average. Doing this many times might finally produce a solution that satisfies our original constraint for every execution. As we are constraining an average value to guarantee a certain maximum value, this might not always succeed though. Depending on the problem, no matter how good the average is, some outliers might still get included; till the point where a even better average is not possible anymore. Therefore, this kind of refinement might fail and also produce suboptimal solutions, because constraining the average is not our real goal.

An alternative is to add additional constraints with each iteration. The idea is that the found solution contains many combinations of which some over-satisfy our QoS constraints and some under-satisfy them. If we can limit the amount of those under-satisfying them, we can increase the probability that the QoS constraints hold for every execution. In order to do so, we can analyze all found combinations, and restrict a weighted sum of the probabilities of services contributing to under-satisfying such a constraint. Lowering the upper bound on this sum and heuristically determining those weights by how often and how much a service contributes to the under-satisfaction should restrict the maximum value of the QoS in question, and give a good utility. Apparently, this assumes two things:

1. Services not selected in prior iterations cannot be selected in later iterations.
2. At least some of the solution's combinations satisfy the QoS constraints.

Regarding the first point, we only want to analyze a hopefully small number of combinations found in the LP problem instead of an exponential number of all

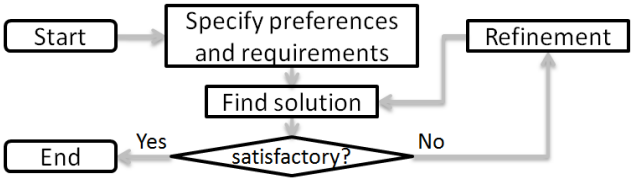


Fig. 8. Refinement Process

combinations possible. As we saw, the latter is not guaranteed, but guaranteeing only a small number of QoS for each execution might still work out quite often and provide better results than the first refinement given a good heuristic for the weights mentioned. Also combining both types of refinements is possible: tighten the corresponding QoS constraint till a satisfying combination is found, and then constrain all under-satisfying combinations as far as needed.

4.4 Computer-Aided Decision Process

The previous applications all had in common that the problem is solved in an automated way. Of course, this is useful in many cases, but it may not be possible or desirable in all cases. For such cases we suggest applying our approach as part of a computer-aided decision process. It has some similarities with the previous application in that the problem gets refined in the process. Only the decision of refining itself and of the kind of refinement is left up to a human. The motivation behind that is the difficulty of defining the optimization problem in terms of the objective function and the constraints. Thus, the idea is to let a human actively control and evaluate the refinement of an initial solution till it fits his needs, instead of forcing him to perfectly specify the objective and constraints in the beginning that would automatically lead to a solution. The decision process could then look similar to Fig. 9 which shows the human activities in grey. In the analysis part e.g. each QoS constraint could be analyzed in depth:

- How often is the constraint kept for each execution?
- How big is the possible deviation from the target value?
- How does the probability distribution of the possible values look like?

A small deviation might for example be acceptable. Interpreting the problem as a multi-objective optimization would also be possible, so instead of computing just one optimal solution, several Pareto-optimal solutions could be computed, each representing different trade-offs. Our approach is particularly suited for such kind of applications, because of its performance: being able to run our computations with different problem settings over and over again in a reasonable time, and, thus, being able to explore different scenarios makes it possible to realize such a computer-aided design process.

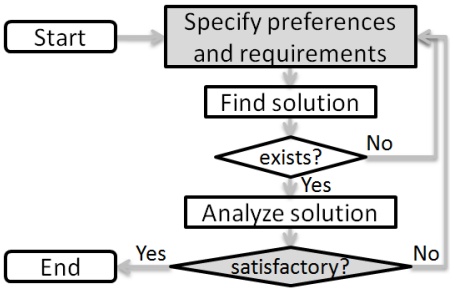


Fig. 9. Decision Process

5 Evaluation

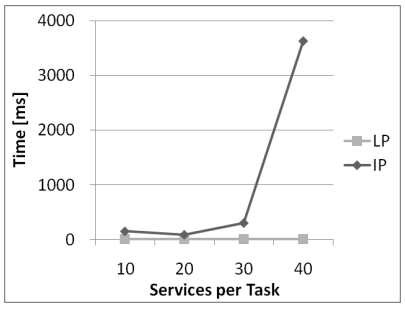
In this section we evaluate our approach taking into account the applications mentioned in the previous section. We evaluate its performance and scalability. Also we analyze the obtained utility gains, and show the results of applying our approach to the original problem.

5.1 Settings

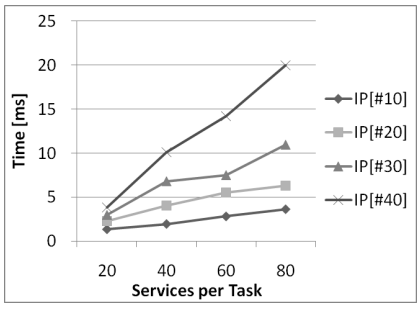
The evaluation was run on a machine with an Intel Core 2 Quad CPU with 3 GHz. As a solver we used CPLEX, a state of the art IP solver from IBM, called from within our Java program which was given 1.5 GB of memory. We generated our workflows as follows: given the workflow size we randomly inserted some control structures, like AND, XOR, et cetera. Unless stated otherwise, the QoS attributes were independently generated with random uniform distributions and typical aggregation patterns. We varied the weights and constraints systematically. The execution time for the optimization was limited to 40 seconds for each individual problem.

5.2 Performance & Scalability

For comparing the performance of LP and IP, we chose 4 attributes with 4 corresponding constraints. For each possible workflow size, 4 different workflows were generated and solved with different weights and constraints for the QoS, resulting in about 250 solved problems per datapoint for both Fig. 10(a) and 10(b). First we compare the performance our LP approach with IP: as shown in Fig. 10(a), the performance is much better for our approach, compared using workflow size 5. We increased the number of services that are available for each task in the workflow from 10 to 40. We can see that already at this small scale using IP at runtime for adaptation or queries is not possible anymore. This is not surprising, because efficient polynomial algorithms are only available for LP.

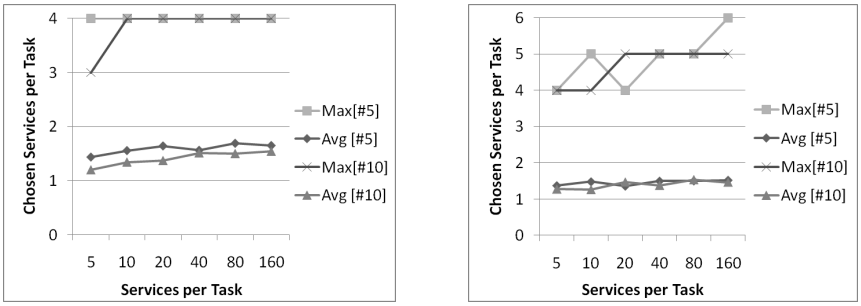


(a) LP vs. IP



(b) LP

Fig. 10. Solution Time



(a) 4 Attributes (b) 8 Attributes

Fig. 11. Chosen Services per Task

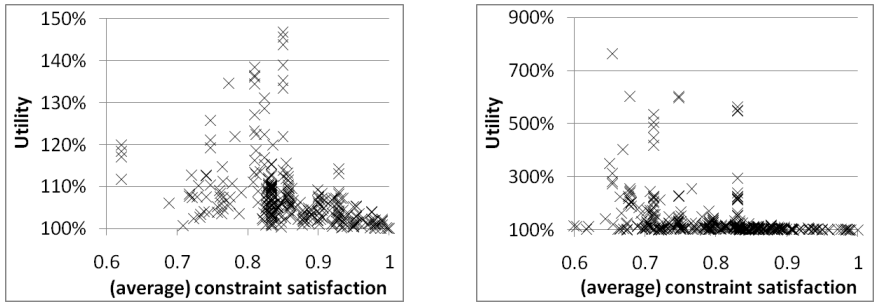
Secondly, we look at the scalability of our LP approach. In Fig. 10(b) we can see the performance of the approach for different workflow sizes (#10–40) and different number of services per task again. The times are below 100ms even for workflows with 40 tasks and 80 services available for each of the tasks. Such performance allows the use even at runtime or under strict performance requirements, e.g. if we want to apply our approach as discussed in the previous section, solving the problem not only once, but several times with different settings.

In addition to this, we also analyze how the number of combinations contained by the solutions found scales with the size of the problem. By comparing Fig. 11(a) and Fig. 11(b) we can observe that the maximum number of chosen services per task for each solution seems to be limited only by the number of attributes. Additionally, there is no significant increase in the the average number of services chosen per task. The observed limit is plausible, because each combination represents a specific trade-off that contributes to an optimal solution, and the number of trade-offs is obviously limited by the number of attributes. Still, the number of combinations per workflow increases exponentially in the worst-case. Hence Monte-Carlo methods will indeed be required to explore the combinations efficiently.

5.3 Utility

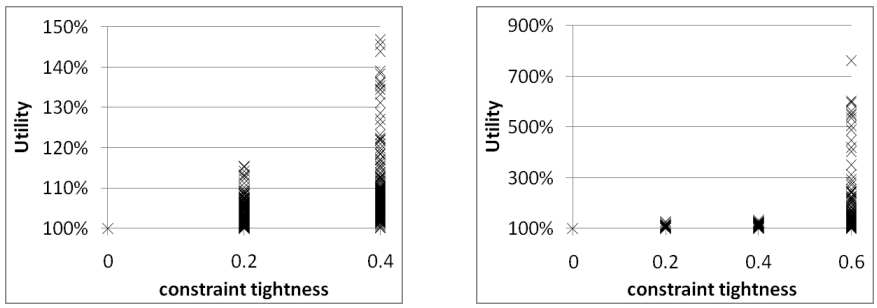
For evaluating the utility obtained by our approach, we chose 6 attributes of which 3 were bounded with constraints. We generated 4 different workflows of size 5 with 5 services available for each task. Then, we solved the problem with some different weights and many different constraints for the QoS, resulting in over 1200 solved problems for each of the following figures. The most important factor that we varied was how to generate the QoS attributes. First, we generated them independently, and, secondly, we generated them assuming a correlation: all QoS attributes x_i were generated independently, except for one attribute x_p which we calculated from the formula[4] $e^{\sum_{i \neq p} x_i}$. Thus, x_p represents a common

[4]Other linear formulas also produced the same tendencies.



(a) Independent QoS (b) Correlated QoS

Fig. 12. Utility vs. Constraint Satisfaction



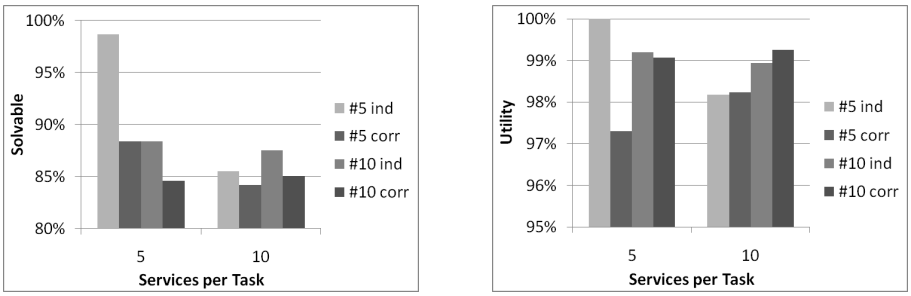
(a) Independent QoS (b) Correlated QoS

Fig. 13. Utility vs. Tightness of Constraints

price attribute, reflecting that services with better QoS are more expensive. As we can see in Fig. 12(a), the utility obtained does not exceed the utility of the IP approach (which corresponds to 100%) by much in most cases given independent QoS, but at the same time the constraint satisfaction is still quite high on average: each constraint is satisfied in at least over 60% of all executions induced by the probabilistic policies. Still there are some cases, where the utility of the composition is improved to 150% of the IP solution. In the second setting with correlated QoS, we see much higher utility gains shown in Fig. 12(b). One reason is obvious: if we want to achieve good QoS, because of the high cost of selecting a single service that is exceptionally good for all QoS attributes, it is cheaper to achieve the same QoS by combining several services that may only be average on most QoS and good for some QoS. Another observation that we can make from Fig. 13(a) and 13(b) is that the tightness of the constraints chosen influences the (potential) utility gains. We define the tightness as a percentage value regarding the location of the constraint between the minimum and maximum QoS possible for a workflow. For loose constraints the solutions of the IP and LP problem converge, and for tighter constraints they diverge greatly.

5.4 Original Problem

We again chose 6 attributes of which 3 were bounded by constraints, and generated 4 different workflows for each size (#5 and #10) with 5 or 10 services available for each task. Then, we solved the problem with different weights and constraints for the QoS which were either independent (ind) or correlated (corr), resulting in over 600 solved problems for each data point in the following figures. From Fig. 14(a) we can see that for each data point a solution of the original problem can be derived in over 80% of the problems. The solution quality in terms of utility achieved is not as good as when solving the IP directly, but, with less than 3% deviation from the optimal IP solution, the approximation quality is very high, as we can see in Fig. 14(b).



(a) Solvability (b) Solution Quality

Fig. 14. Solving the Original Problem

6 Conclusion

In this paper we introduced an efficient approach for the composition problem that optimizes QoS for repeated executions of services in the long-term. Our approach produces a set of probabilistic service selection policies that must be evaluated at runtime to choose a particular service accordingly. We demonstrated several potential applications of our approach beyond the mentioned scenario, including a computer-aided decision process. We evaluated our approach and showed that it is indeed efficient and scalable for the needs envisioned in SOA. We also showed in which cases our approach yields the most utility gains over common approaches. Additionally, we showed that our approach can be used to solve the traditional composition problem in many cases and produces near-optimal solutions.

Exploring the various applications of our approach in depth, and developing corresponding methodologies and tools are possible extensions of our work. Using our approach in conjunction with a genetic algorithm to compute a multi-objective optimization yielding multiple Pareto-optimal solutions for different trade-offs would also be a possible extension of our work.

7 Acknowledgments

We thank Nobuaki Hiratsuka for his work using multiple services that challenged our assumption to just select a single service for each workflow element. We also like to thank Florian Wagner and Benjamin Klöpper for the fruitful discussions that helped form our approach.

References

1. Papazoglou, M. P., Traverso, P., Dustdar, S., Leymann, F., Kramer, B.J.: Service-oriented computing: A research roadmap. In: Service Oriented Computing (SOC). Dagstuhl Seminar Proceedings (2006)
2. O'Sullivan, J., Edmond, D., Ter Hofstede, A.: What's in a Service? Distributed and Parallel Databases 12(2–3), 117–133 (2002)
3. Chen, K., Xu, J., Reiff-Marganiec, S.: Markov-HTN Planning Approach to Enhance Flexibility of Automatic Web Service Composition. ICWS '09: IEEE International Conference on Web Services, 9–16 (2009)
4. OASIS Committee Draft: Web Service - Business Process Execution Language (WS BPEL), Version 2.0. (2006)
5. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality Driven Web Services Composition. In: WWW '03: Proceedings of the 12th international conference on World Wide Web, 411–421 (2003)
6. Yu, T., Zhang, Y., Lin, K.-J.: Efficient algorithms for Web services selection with end-to-end QoS constraints. ACM Transactions on the Web 1(1), 6 (2007)
7. Pisinger, D.: Algorithms for Knapsack Problems. PhD thesis, University of Copenhagen, Dept. of Computer Science (1995)
8. Menascé, D.A., Casalicchio, E., Dubey, V.: On optimal service selection in Service Oriented Architectures. Performance Evaluation 67(8), 659–675 (2009)
9. Lecue, F., Mehandjiev, N.: Towards Scalability of Quality Driven SemanticWeb Service Composition. In: ICWS '09: IEEE International Conference on Web Services, 469–476 (2009)
10. Canfora, G., Di Penta, M., Esposito, R., Villani M.L.: An approach for QoS-aware service composition based on genetic algorithms. In: GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, 1069-1075 (2005)
11. Alrifai, M., Risse, T.: Combining global optimization with local selection for efficient QoS-aware service composition. In: WWW '09: Proceedings of the 18th international conference on World wide web, 881–890 (2009)
12. Alrifai, M., Skoutas, D., Risse, T. : Selecting Skyline Services for QoS-based Web Service Composition. In: WWW '10: Proceedings of the 19th international conference on World wide web, 11–20 (2010)
13. Stein, S., Payne, T. R., Jennings N.R.: Flexible provisioning of web service workflows. ACM Transactions on Internet Technology 9(1), 1–45 (2009)
14. Klein, A., Ishikawa, F., Bauer, B.: A Probabilistic Approach to Service Selection with Conditional Contracts and Usage Patterns. In: ICSOC-ServiceWave '09: Proceedings of the 7th International Joint Conference on Service-Oriented Computing, 253–268 (2009)
15. Jaeger, M.C., Rojec-Goldmann, G., Muhl, G.: QoS Aggregation for Web Service Composition using Workflow Patterns. In: EDOC '04: Proceedings of the Eighth IEEE International Enterprise Distributed Object Computing Conference, 149–159 (2004)