A Probabilistic Approach for Long-Term B2B Service Compositions

Adrian Klein, Florian Wagner
University of Tokyo
Tokyo, Japan
{*adrian,florian*}*@nii.ac.jp*

Fuyuki Ishikawa, Shinichi Honiden
National Institute of Informatics
Tokyo, Japan
{*f-ishikawa, honiden*}*@nii.ac.jp*

***Abstract*—Service composition algorithms are used for realizing loosely coupled interactions in Service-Oriented Computing. Starting from an abstract workflow, concrete services are matched, based on their QoS, with the preferences and constraints of users. Current approaches usually only consider static QoS values and find a single solution consisting of one concrete service for each workflow task. In a business-to-business (B2B) environment, though, there are additional requirements for service compositions: 1) a high number of invocations, and 2) a high reliability. Thus, we introduce a probabilistic approach on the basis of a new QoS model to solve the composition problem for such long-term B2B service compositions. For each task and for every point in time, we determine the most appropriate services and backup services for a specific user. Thus, the selection depends on the actual response time and reliability, or recent invocation failures or timeouts. For that purpose, we propose an adaptive genetic algorithm that employs our QoS model and determines backup services dynamically based on the required reliability. Our evaluations show that our approach significantly increases the utility of long-term compositions compared with standard approaches in the envisioned B2B environments.**

***Keywords*-QoS-aware service composition, long-term, B2B, reliability**

I. Introduction

Service-Oriented Computing is based on the principle of combining loosely coupled services in order to achieve complex functionality. Abstract workflows (cf. Fig. 1) capture the business logic of such services compositions. Service selection algorithms determine concrete services for each workflow task, which offer the necessary functionality.

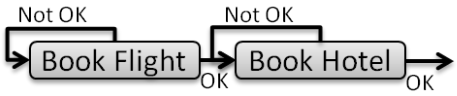


Figure 1: Abstract Workflow with two Tasks

A. QoS-aware Service Composition

Service providers specify not just the functionality of their services, but also their non-functional Quality of Service (QoS) properties. Thus, following after the functional selection, the service composition problem deals with choosing a good service in terms of QoS for each task. Given this selection, the QoS of all services are aggregated to compute the QoS of the whole composition. Users state their preferences towards the QoS in form of a utility function and constraints on the acceptable QoS values. Since finding an optimal combination of services is a NP-complete problem, heuristic algorithms are used to find near-optimal solutions.

Most current composition approaches focus on finding solutions for a single invocation of a workflow. Accordingly, the QoS of services are considered as static values that are guaranteed by their providers. Thus, for each new invocation, either the previous solution is reused, or, if the QoS change, a new solution has to be computed from scratch.

B. B2B Service Compositions

In contrast to traditional service composition, we propose a different approach that is tailored to service compositions in a business environment. As more and more companies migrate their services to the cloud and advance the integration with services from other companies, such business-to-business (B2B) environments become more important for service compositions. We envision the two following major requirements for such B2B service compositions.

1) Long-Term: First of all, B2B service compositions are mainly intended for long-term usage. In the following, we illustrate the three main reasons for this in the order of the natural flow, when using B2B service compositions.

- **Building** B2B workflows takes effort, as complex business functionality has to be captured in a correct and complete way, e.g. dealing with multiple possible paths, error handling, etc.
- **Integrating** B2B workflows requires a considerable effort in customization, testing, and verification. Algorithms can automatically find concrete services, which can be used to execute an abstract workflow. However, in practice, the correct passing and formatting of parameters might need to be customized. Moreover, the correctness and the format of the returned results should be tested and verified. Apart from that, actual contracts might have to be signed between the business parties involved, and privacy policies might have to be reviewed by lawyers, etc.
- **Executing** B2B workflows is an ongoing task, requiring constant monitoring and resulting in many repeated and possible concurrent invocations over a long time frame.

2) Reliability: Apart from considering the long-term utility, B2B service compositions require a very high reliability. Usually, reliability is measured by the number of 9s, e.g. a reliability of 99% corresponds to two 9s, etc. The number of desired 9s depends on the type of application. Table I shows reliability requirements for different application types according to the Standish Group [1, Table 1-3].

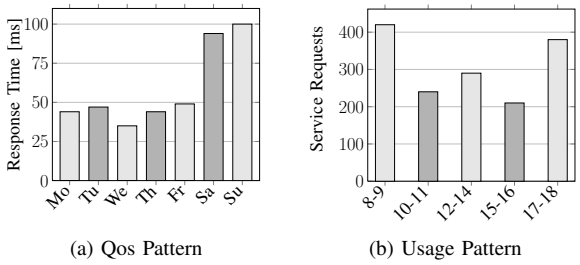
Application type	9s	%
Non-critical	2	99%
Task critical	3	99.9%
Business critical	4	99.99%
Mission critical	5	99.999%
Safety critical	6	99.9999%

Table I: Reliability Requirements (The Standish Group)

C. Probabilistic Approach

We propose a probabilistic approach in order to handle B2B service compositions that are executed over the long-term and require a high reliability. In the following, we introduce the four major concepts we base our approach on, dealing with different aspects of the previous requirements.

QoS Patterns: The QoS of services are not static over the long-term. Thus, we model their QoS as arbitrary patterns that are repeated over time. For example, the response time of a service for booking a hotel room might depend on the day of the week, due to an increased load on the weekend (cf. Fig. 2a). These QoS patterns can be computed from the history of previous service invocations.



(a) Qos Pattern (b) Usage Pattern

Figure 2: Examples of a QoS and Usage Pattern

Usage Patterns: The service invocations from business users usually follow a pattern over the long-term as well. For example, a service for determining physical access control could show a usage pattern as in Fig. 2b, peaking when most employees either enter or leave the company.

Time-dependent Execution Policy: To take both QoS and usage patterns into account, we solve the composition problem not just for a single point in time, but concurrently for a finite number of points in time, which are determined by the available patterns. Therefore, our computed solution is a time-dependent execution policy that determines the best concrete services for each point in time.

Backup Services: We achieve the required high reliability by selecting multiple services for each task. This selection consists of one primary service and several backup services that are invoked if the primary service times out. For example, it would be very difficult to achieve an reliability of four 9s for a workflow with 40 services by selecting a single service for each task; each service would need an reliability of more than five 9s (i.e. more than mission critical). Contrary to that, we can easily achieve that reliability by selecting three services for each task, each with an reliability of two 9s. Apart from that, selecting multiple services for a task is usually cheaper than using services with higher reliability that are both more rare and more expensive.

D. Contributions

The contributions of this paper are as follows:

1) We motivate and formalize the service composition problem for long-term B2B service compositions.
2) We propose a holistic probabilistic approach using QoS patterns, usage patterns, and time-dependent invocation policies. In this way, we can model the QoS of each service, the user's usage of each service, and the invocation for each point in time.
3) We introduce our adaptive heuristic algorithm, Teikou, that employs this QoS model. It is based on a genetic algorithm and adjusts the number of backup services to the reliability constraint of the user.
4) Through our evaluations we show that, for B2B service compositions, our approach greatly improves both the long-term utility and the reliability, compared to current approaches.

The structure of the following paper is as follows. First, we describe our related work in Sect. II. Then, we define the preliminaries and assumptions of our approach in Sect. III. In Sect. IV we introduce our QoS model and present our adaptive algorithm in Section V. We evaluate our approach in Sect. VI, and draw a conclusion in Sect. VII.

II. Related work

A. QoS-aware Service Composition

Our approach is based on the QoS-aware service composition problem defined in [2]. Extensive rules for aggregating the QoS of a service composition have been described by [3]. Efficient algorithms have been introduced first in [4] and most recently in [5]–[7]. We share the common notations defined in these papers.

B. Complex QoS

We assume that the QoS of services are complex following patterns/distributions over time, rather than being defined as static values. This means that we treat them in a probabilistic manner, as in [8], [9]. QoS patterns for providers were also applied by us in [10], [11]; later Chen et al. [12] considered how to find the time cycles of such patterns.

C. Long-term Service Compositions

In order to account for the long-term, we consider usage patterns which we first used in our previous work [10]. A probabilistic selection policy for the long-term was introduced by us in [13], but this policy was not time-dependent and did not consider backup services with an execution order, as to allow to compensate service failures.

D. Reliable Compositions

Backup services [14] or replanning [15] can be used to compensate a service failure. However, in this study we consider the long-term utility of services that have fixed (contract) costs in addition to the usual costs per invocation. Therefore, the number of backup services has to be determined before invoking the workflow. Jaeger et al. present in [16] an approach that calls a static number of services in parallel to increase the reliability. This approach is not applicable in our problem setting, since services have also fixed contract costs. In [17], a planning algorithm is presented that clusters functionally related services into groups. In contrast to service selection, this approach generates workflows automatically. This approach cannot consider constraints on the QoS and moreover applies a simplified QoS model. In [18], we have shown a QoS model to compute the value of each QoS attribute in the best-, worst-, and expected-case. However, this model increases the complexity of the selection significantly, since three values are computed for each QoS attribute. Moreover, the expected-case is rather pessimistic since no reimbursements are granted, whereas in this scenario service users sign contracts with the providers. In addition, we also optimize the order in which the services would be called in case of service failures.

III. Assumptions

In this section, we discuss preliminaries and assumptions of this paper regarding the common QoS-aware service composition problem.

A. Services and QoS

A *service* encapsulates a certain business functionality. The input and output parameters of a service are defined in service description documents. These documents also declare the Quality-of-Service (QoS) attributes of the service, such as the price and response time. Moreover, each service has a reliability, indicating its success rate. Data about the reliability is usually collected from execution logs to facilitate forecasts [19] of the success rate.

Services with the same input and output parameters are accumulated in a *service task*, sharing the same purpose, but offering varying QoS levels. An *abstract workflow* defines the dataflow of the service tasks through sequences, and/or branches, and loops.

B. QoS-aware Service Composition

QoS-aware composition algorithms refine abstract workflows to *executable workflows* by selecting for each task one service. The set of chosen services is also called a *service selection*. On the basis of a service selection, the resulting QoS of the executable workflow can be computed, e.g. by the computation rules provided in [20, Table 1].

The user may specify certain constraints that limit the set of feasible selections, such as the maximum price or minimum reliability. In order to further emphasize a QoS, he/she can express preferences towards certain QoS attributes by specifying their weights for a utility function μ:

$$\mu(W) = \sum_{i=1}^{|W.Q|} w_i \cdot W.Q_i$$

Optimizing the QoS while meeting the constraints is a NP-hard problem [2]. For that reason, heuristic algorithms like hill-climbing algorithms or genetic algorithms (GA) are used to find near-optimal solutions. Most of such heuristic algorithms use a fitness function to evaluate how good/optimal a potential solution is. This fitness function usually is defined as the utility of the corresponding workflow W minus a penalty indicating the distance to the constraints (if they are not met):

$$\mu_{fit}(W) = \mu(W) - \sum_{i=1}^{|C|}(W.Q_i - C_i)$$

By using such a fitness function, heuristic algorithms are guided to solutions that meet the constraints and have good QoS.

IV. QoS Model for Long-term Utility

In this section, we introduce our QoS model for the computation of the utility over the long-term. First, we give definitions of our QoS and usage patterns. Based on these definitions, we specify the time-dependent execution policy obtained by our approach. Finally, we introduce our concepts for the long-term utility and define how to compute it.

A. QoS Patterns

While most current approaches consider QoS to be static values, we define them in a time-dependent fashion. We assume that the QoS of services follow some kind of pattern, e.g. over the timespan t_{week} of a week. Thus, there exists some finite collection of sets $t_{pattern} = \{tp_1, tp_2, ..., tp_p\}$ such that the following holds:

$$\bigcup_{i=1}^{p} tp_i = t_{week}$$

$$\forall_{i,j}.i \neq j \Rightarrow tp_i \cap tp_j = \oslash$$

Then, we can specify the QoS of services by defining functions $f_{Q_i} : t_{pattern} \rightarrow [0,1]$ that map each element, tp_i, of the pattern to different values for each QoS Q_i.

B. Usage Pattern

A usage pattern can be interpreted as the distribution of the user's usage of a service over a certain timespan, e.g. over a week t_{week}. This results in a collection of sets $t_{usage} = \{tu_1, tu_2, ..., tu_u\}$ and a single function $u : t_{usage} \rightarrow [0,1]$ that computes the percentage of the expected usage, which occurs at a certain time tu_i. Thus, these numbers add up to 100%.

C. Time-Dependent Execution Policy

In order to take both the QoS and the usage patterns into account, the solutions obtained by our approach contain not just a single concrete service, but a *time-dependent execution policy* for each task. Furthermore, since services might crash, backup services are employed to compensate service failures. Therefore, for each point in time, tu_i, the policy provides a *service group*. A service group is an ordered set of services with the first service being the primary service. This service will be called first; the other services are backup services that will be called according to their order in case of service failures of their preceding services. This way, we can select the most appropriate service at any point in time without additional computational cost even when some services fail. Increasing the number of backup services increases the reliability of a task but also adds fixed contract costs. Note that, for the long-term, the user often commits to pay a fixed contract fee in current B2B environments, see Amazon's EC2's pricing [1]. For that reason, the optimal number of backup services depends on the required reliability of a workflow and the fixed costs.

D. Long-term Utility

We optimize the utility over the long-term for our envisioned B2B environments. We realize this by optimizing for a given number of expected invocations, e.g. 100 invocations per month. Either this number is specified by the user, or it is derived from the invocation history. Using the services' QoS patterns and the user's usage pattern, we then evaluate the expected QoS values of a service composition over the long-term. Note that, while we accumulate the utility over the long-term, we normalize it by computing the expected utility for a single invocation.

*1) **Computing the Reliability**:* A service group $G \subseteq T$ is a ordered subset of the services assigned to a task T. In order to compute the reliability $G.rel$ of a service group, we consider the reliability $S.rel$ of each service in the group:

$$G.rel := 1 - \prod_{S \in G} \big(1 - S.rel\big)$$

[1] http://aws.amazon.com/ec2/pricing/

The success probability of executing a group is the probability that at least one service can be executed successfully. By grouping services we can improve the reliability of a task significantly. In our model, the reliability of a service is considered to be independent of other services.

Example: Consider a group G of three services S_1, S_2, and S_3, having the reliability values $\{0.5, 0.2, 0.3\}$. In total, group G has reliability:

$$G.rel = 1 - (1 - 0.5) \cdot (1 - 0.2) \cdot (1 - 0.3) = 72\%$$

*2) **Computing Price and Response Time**:* The price and response time of service groups depends on the QoS and the number of invocations of each contained service. As groups determine an invocation order beforehand, this number depends on the previous services and their reliability.

Each user provides the number of expected invocations C of a workflow beforehand. On the basis of that, we can compute the number $calls_i$ of successful invocations of the first service of a group by:

$$calls_0 := C \cdot S_0.rel$$

For the other services in the group we apply the following:

$$calls_i := \big(C - \sum_{j=0}^{i-1} calls_j\big) \cdot S_i.rel$$

Consider the previous example with three services and 100 invocations. If we execute the services in ascending order, the first service is invoked on average 50 times, the second 10 times, and the last 12 times. Therefore, the workflow is successfully executed for 72 times. We compute the price and response time of each service as:

$$G.price := \sum_{i=0}^{|G|-1} \big(S_i.price \cdot \lceil calls_i \rceil + contractCosts_i\big)$$

The response time is computed in a similar way, but without the fixed contract costs. After computing the QoS of the service groups, we aggregate the QoS of the workflow as discussed in Section III-B.

V. Adaptive Genetic Algorithm

In this section, we present our Teikou algorithm which makes use of our previously introduced QoS model and computation. The Teikou algorithm adapts the sizes of the services groups dynamically, because the size of the service groups depends on the reliability constraints and the QoS of the services. Moreover, this adaptation is done per task, as the average reliability of services may vary greatly per task.

As a basic framework we employ a Genetic Algorithm (GA) that uses a genome encoding with a variable size to adjust the service groups. In the beginning, our heuristic estimates the initial sizes of the groups. During the evolution of the genomes, an *adapt* operation is applied to the genomes. If the reliability of the workflow is too low, the

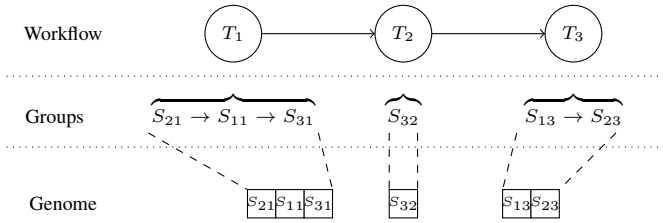


Figure 3: Encoding of a Service Group Selection

groups with the lowest reliability are enlarged. On the other hand, if the reliability is sufficient, but the costs are too high, the operation shrinks the groups with the highest reliability.

First, we explain how the variable genome encoding is realized. After that, we discuss how to determine initial group sizes and how to adjust existing mutate and crossover operations. Finally, we specify our *adapt* operation.

A. Problem Encoding

We use the common encoding of the service selection problem [20] as a basis. The index of the services of the groups are the cells of the genome. The genome length is adjusted to the sizes of the groups. Each cell keeps track of its corresponding service group. This information is needed by the mutate and crossover operations. Figure 3 illustrates the genome encoding of a workflow.

B. Group Size

During the evolution process we adapt the sizes of the service groups, enlarging or shrinking them to match the reliability constraint. Furthermore, determining good initial group sizes is crucial to achieve a fast convergence of the GA. For those two purposes, we determine the minimum and maximum desired reliability of a service task. Using a too low reliability will result in violating the reliability constraint, whereas using a too high reliability unnecessarily increases the cost of the workflow. These two values are used to compute ranges of the desired group sizes. First, we compute the range of the desired reliabilities for all tasks:

$$minrel := \sqrt[c_{min}]{relconst} \qquad c_{min} := \frac{wfsize}{\epsilon} \quad (1)$$

$$maxrel := \sqrt[c_{max}]{relconst} \qquad c_{max} := wfsize \cdot \epsilon \quad (2)$$

Next, for each task T, we compute the average reliability of all its services:

$$avgRel_T := \frac{\sum_{S \in T} S.rel}{|T|}$$

In our experiments, we used $\epsilon = 2$. Note that increasing ϵ allows for more possible solutions to be explored, while decreasing it speeds up convergence. In the final step, we compute the minimum and maximum size of the service group for each task T:

$$minsize_T := \lfloor \log_{1-avgRel_T}(1 - minrel) \rfloor$$

The value $maxsize_T$ is computed in the same way with $maxrel$. The initial size of a service group is then computed by using a random value from the range $[minsize_T, maxsize_T]$. Considering the previous example, if we apply a reliability constraint of 20% on a workflow of length 3, the heuristic would generate service groups with sizes between 1 and 3. We also use the desired reliabilities for our adapt operation which we introduce later.

C. Genetic Operators

After generating an initial population, Teikou performs the three standard operators of a GA plus an additional adapt operator. However, since we employ service groups instead of single services, we adjust the standard operators.

1) ***Mutate Operator****:* The value of a cell is replaced with a new value with probability P_{mut}. When selecting a cell, the group of that cell is retrieved. Only values that are not used in the group are eligible for replacing the old value.

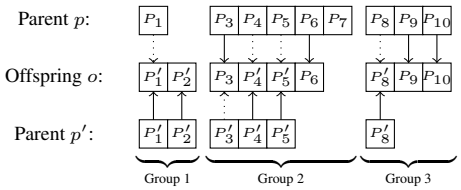


Figure 4: Using the Uniform Crossover on two Individuals with varying Genome Lengths

2) ***Crossover Operator****:* A genome G is crossed with a random partner G' to generate a new offspring genome C with probability P_X. Several crossover operators exist in literature; in the following, we apply the uniform crossover. For each cell C_i, this operator picks either G_i or G'_i of the parent genomes with equal probability. However, individuals with varying genome lengths might be selected, as shown in Fig. 4. The resulting group size of the offspring is from the range

$$\Big[\min\big(|G|, |G'|\big), \max\big(|G|, |G'|\big)\Big]$$

In case that the offspring has more services in a specific group than one parent, the cells from the other parent are used. In few cases, the GA might select a value for cell C_i where both values from parents G_i and G'_i are already contained in the previous cells $\{C_0, \ldots, C_{i-1}\}$. In that case, none of the parents' cells can be used and the GA performs the crossover operation for this group again from the start.

*3) **Select Operator**:* In the end of a generation, the genomes with the highest fitness values are selected and used for the next generation. After that, this process is iterated either a fixed number of times or until the gain of the last n iterations is less than a threshold ϵ.

*4) **Adapt Operator**:* The adapt operation is executed after applying the mutate and crossover operations. In the beginning, we determine the difference between the desired and the actual reliability of each service group. On this basis, we rank the groups of each genome and adapt a fixed ratio P_{adapt} of them.

Algorithm 1: rankGroups

Input: Genome G
Output: Priority queue PQ, containing groups with wrong group sizes

```
1 foreach Group g ∈ G do
2     Δ := g.rel − ^|W|√relconst;
3     if |Δ| ≤ τ then continue;
4     if Δ < 0 ∧ |g| ≥ maxSize then continue;
5     if Δ > 0 ∧ |g| = 1 then continue;
6     g.Δ := Δ;
7     PQ.push(g);
8 end
9 adaptGroup(PQ);
```

Algorithm 1 is the pseudo code of our genome ranking. In the beginning, we iterate over the service groups of G. In line 2, we compute the difference Δ between the reliability of group g and the desired reliability of that task. If Δ is smaller than a threshold τ we omit this group in order to avoid unnecessary modifications that hamper the convergence of the GA. In lines 4 and 5, we check whether we can apply the adapt operation. If the group has only one member or already the maximum number of members, we cannot apply a shrink or enlarge operation, respectively. If we can apply an adapt operation on this group, we keep the Δ and add the group to the priority queue PQ.

After ranking the groups of a genome, we perform the adapt operation on these groups, shown in Algorithm 2. In the beginning, we compute the number of service groups we will adapt and clone the original genome (lines 1 and 2). After that, we retrieve the top element of the priority queue and check whether we have to apply a shrink or enlarge step. If the reliability of the group is too low, we add a random service from task t that is not yet contained in g' in line 8. In the other case, we remove the last element of g' in line 9. In the end, we replace the original group g by g' and iterate the process. Figure 5 illustrates the effect of enlarging a service group on the genome encoding.

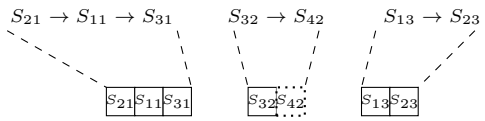


Figure 5: Genome Encoding after Enlarging a Service Group

Algorithm 2: adaptGroups

Input: Priority queue PQ
Output: Modified genome G'

```
1  count := ⌈P_adapt · |W|⌉;
2  G' := clone(G);
3  while count > 0 ∧ PQ ≠ ∅ do
4      g := PQ.poll;
5      g' := clone(g);
6      if g.Δ < 0 then
7          Service s := random(T \ G');
8          g' := g' ∪ s;
9      else g' := g' \ {last(g')};
10     G'.replace(g, g');
11     count := count - 1;
12 end
```

VI. Evaluation

In this section we evaluate our approach. First, we explain the setup and settings of our approach. Then, we compare the utility achieved for users against an increasing reliability constraint, which makes it more difficult to find possible solutions. This way, we can evaluate the benefits of our adaptive approach and of our usage patterns. After that, we evaluate the reliability and runtime against an increasing workflow size to show the scalability of our approach.

A. Setup

Implementation. Our implementation is written in Java. We ran our experiments on a machine with 3.00 GHz; we assigned a maximum memory of 2 GByte to the Java VM. The GAs used 100 genomes, P_{mut} was 5% and P_X was 70%. Each GA terminated if either 150 generations were performed or if the utility of the best genome did not increases by at least 1% in the last ten iterations.

Problem Setting. For each data point in the following graphs, we ran 100 different test cases with randomly generated services, workflows, and QoS values. Services had a reliability between 90% and 99.999%, and a price that is anti-correlated to the reliability. Each task has 50 services and an QoS influence that slightly shifts the QoS of the contained services. Each workflow consists of ten tasks, connected by sequences, and-branches, and or-branches. The utility is composed of only the price in our evaluations, in order to be anti-correlated to the reliability constraint.

Ideal Utility. Determining the optimal solution in our problem setting is not possible in feasible time. Consider a workflow with ten tasks and 50 services. The solution

space for single service selection contains 50^{10} solution candidates, without even considering QoS and usage patterns. However, since the algorithms can select a service group with up to six services for each task, a lower bound for the solution space is

$$\left(\frac{50!}{44!}\right)^{10}$$

Therefore, we calculate an *"ideal" solution* for each data point: for each task, we take the best (cheapest) service group from all computed solutions, and then aggregate the prices of these groups to find the price of an *ideal solution* for the whole workflow. Note that this solution may violate the reliability constraint, but still is an indicator for the best possible utility in the given problem setting.

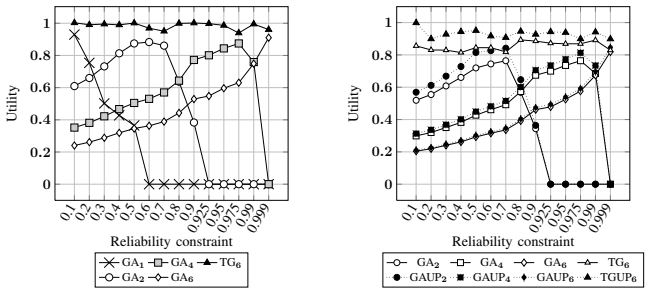
Constraints We vary the reliability constraint from 10% (very easy to solve, but not applicable) up to 99.9% (corresponds to "task critical" operations, cf. Section I). Please note that we evaluate constraints lower than 90% only to illustrate the behaviour of the algorithms. Workflows with such low reliability are not desirable in reality. If a solution violates the constraint, its utility becomes zero.

B. Utility

We evaluated several algorithms with and without consideration of usage patterns in the evaluation of the utility of our approach, which is plotted in Figure 7.

$\boldsymbol{GA_i}$ denotes a naïve adoption of using service groups with a static group size that is denoted by the index.

$\boldsymbol{TG_6}$ denotes our adaptive Teikou algorithm which determines the groups sizes dynamically up to a maximum size of six services per group.



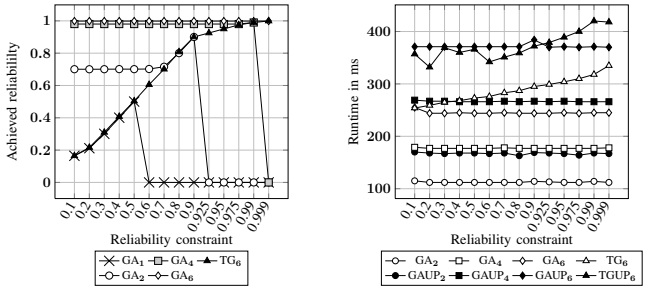
(a) Utility without Usage Patterns (b) Utility with Usage Patterns

Figure 6: Comparing the Utility against Varying Constraints

In the first experiment, shown in Figure 6a, we evaluate how Teikou performs in a scenario without usage patterns. We see that all GAs with static group sizes show the same pattern; first, with increasing reliability constraints their utility comes closer to the ideal solution. After that point is reached, their utility drops quickly. This means that for each groups size, an ideal reliability constraint exists. Our adaptive approach performs well for all constraints, achieving over 90% of the utility of the ideal solution.

Figure 6b shows the second experiment. In this setting, the user has a random usage pattern. For each weekday, we generate a probability $\boldsymbol{P_w}$ that indicates how often the user executes the service on that day. Moreover, each service has a time-dependent pricing model with a random discount on the price depending on the weekday. We use a GA with group sizes two, four, and six. Algorithms with an "UP" at end recognize usage patterns. In this way, we can compare the effect of considering usage patterns.

Similar to the first experiment, with increasing reliability constraints the utility increases until a certain point for GA with static group sizes. Near that point, the difference between employing usage patterns and neglecting them becomes more evident. Teikou achieves the best results regardless of the reliability constraints. Only the version without usage patterns is sometimes outperformed by static GAs that are allowed to employ usage patterns.



(a) Reliablilty (b) Runtime

Figure 7: Comparing Reliability and Runtime Performance

C. Reliability

In the next experiment (cf.Fig. 7a), we compare the reliability of the group selections. Each algorithm can adjust itself to the reliability constraint to a certain extent. For instance, $\boldsymbol{GA_1}$ achieves exactly the required reliability in the range of $[\mathbf{0.1}, \mathbf{0.5}]$. However, after that $\boldsymbol{GA_1}$ violates the constraint. The results confirm that the Teikou algorithm can adapt to the reliability constraint. Furthermore, the intuition is that by achieving exactly the required reliability, a higher utility can be achieved in a situation when the constraint is tight and the utility and the constraint are anti-correlated. So this result also explains why Teikou can achieve a higher utility than the other GAs.

D. Runtime

In the last experiment, we compare the runtime of each algorithm, shown in Figure 7b. The runtime of each algorithm is apparently independent of the reliability constraint, except for Teikou; with increasing constraint, the runtime increases. Since Teikou enlarges the genomes as the reliability constraint increases, the computation time increases as well. However, the runtime increases only slightly compared

with related GAs. Moreover, the runtime of algorithms that consider usage pattern increases by a constant factor.

E. Summary

We conclude that Teikou achieves the best results in terms of utility and reliability regardless of the reliability constraint. The runtime increases linearly with increasing problem complexity. Employing usage patterns increases the utility as well, but worsens the runtime slightly. In the B2B environments we envision though, Teikou would be applied to optimize the utility in the long-term resulting in time-dependent execution policies. Thus, for online settings, considering usage patterns or not is a trade-off between utility and runtime, but, for long-term decisions, usage patterns definitely provide a big benefit.

VII. Conclusion

In this paper, we have presented the problem of long-term B2B service compositions. Therefore, we introduced a new QoS model that allows to compute the QoS of a workflow for the long-term by taking service groups into account. Our algorithm Teikou employs a genetic algorithm with modified standard operators and a new *adapt* operation that adjusts the sizes of the groups dynamically to achieve the user's reliability constraint. We compared our algorithm with various GAs using static groups sizes and showed that our adaptive algorithm Teikou achieves a higher utility. We also showed that employing usage patterns further improves the utility and that the trade-off versus the increased runtime is negligible for the B2B environments we envision.

As future work, we plan to investigate realistic usage patterns and QoS patterns by analyzing existing service execution logs. In this paper, we made certain assumptions towards the pricing models for the sake of simplicity; as a next step we intend to explore a wider range of pricing models as well.

Acknowledgements

Adrian Klein is supported by a Research Fellowship for Young Scientists from the Japan Society for the Promotion of Science. The work of Florian Wagner is partially supported by the KDDI Corporation.

References

[1] B. Highleyman, *Breaking the Availability Barrier: Survivable Systems for Enterprise Computing*, 2003.

[2] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng, "Quality Driven Web Services Composition," in *Proceedings of the 12th International Conference on World Wide Web (WWW)*, 2003, pp. 411–421.

[3] M. Jaeger, G. Rojec-Goldmann, and G. Mühl, "QoS Aggregation for Web Service Composition using Workflow Patterns," in *Proceedings of the Eighth IEEE International Enterprise Distributed Object Computing Conference*, 2004.

[4] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient Algorithms for Web Services Selection with End-to-End QoS Constraints," *ACM Transactions on the Web*, vol. 1, no. 1, p. 6, 2007.

[5] M. Alrifai and T. Risse, "Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition," in *Proceedings of the 18th International Conference on World wide web (WWW)*, 2009, pp. 881–890.

[6] 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 (WWW)*, 2010, pp. 11–20.

[7] A. Klein, Adrian, F. Ishikawa, and S. Honiden, "Efficient Heuristic Approach with Improved Time Complexity for QoS-aware Service Composition," in *The 9th International Conference on Web Services (ICWS)*, 2011.

[8] S. H. S. Rosario, A. Benveniste and C. Jard, "Probabilistic QoS and Soft Contracts for Transaction-Based Web Services Orchestrations." vol. 1, no. 4, 2008, pp. 187–200.

[9] H. Zheng, J. Yang, and W. Zhao, "QoS Analysis for Web Service Compositions Based on Probabilistic QoS," *International Conference on Service-Oriented Computing (ICSOC)*, pp. 47–61, 2011.

[10] A. Klein, F. Ishikawa, and B. Bauer, "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*, 2009, pp. 253–268.

[11] B. Kloepper, F. Ishikawa, and S. Honiden, "Service Composition with Pareto-Optimality of Time-Dependent QoS Attributes," in *International Conference on Service-Oriented Computing (ICSOC)*, ser. Lecture Notes in Computer Science, vol. 6470, 2010, pp. 635–640.

[12] L. Chen, J. Yang, and L. Zhang, "Time Based QoS Modeling and Prediction," *International Conference on Service-Oriented Computing (ICSOC)*, pp. 532–540, 2011.

[13] A. Klein, F. Ishikawa, and S. Honiden, "Efficient QoS-Aware Service Composition with a Probabilistic Service Selection Policy," in *Proceedings of the International Conference on Service Oriented Computing (ICSOC)*, 2010, pp. 182–196.

[14] M. Laukkanen and H. Helin, "Composing Workflows of Semantic Web Services," in *Proceedings of the Workshop on Web-Services and Agent-based Engineering*, 2003.

[15] K.-J. Lin, J. Zhang, and Y. Zhai, "An Efficient Approach for Service Process Reconfiguration in SOA with End-to-End QoS Constraints," in *CEC*, B. Hofreiter and H. Werthner, Eds. IEEE Computer Society, 2009, pp. 146–153.

[16] M. C. Jaeger and H. Ladner, "Improving the QoS of WS Compositions Based on Redundant Services," in *Proceedings of the International Conference on Next Generation Web Services Practices*, ser. NWESP '05. IEEE Computer Society, 2005.

[17] F. Wagner, F. Ishikawa, and S. Honiden, "QoS-Aware Automatic Service Composition by Applying Functional Clustering," *IEEE International Conference on Web Services (ICWS)*, 2011.

[18] F. Wagner, B. Klöpper, F. Ishikawa, and S. Honiden, "Towards Robust Service Compositions in the Context of Functionally Diverse Services (to appear)," in *21st International Conference on World Wide Web (WWW)*, 2012.

[19] A. G. Liu, E. Musial, and M.-H. Chen, "Progressive Reliability Forecasting of Service-Oriented Software," *IEEE International Conference on Web Services (ICWS)*, vol. 0, pp. 532–539, 2011.

[20] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, "An Approach for QoS-aware Service Composition based on Genetic Algorithms," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO)*, 2005, pp. 1069–1075.