Metaheuristic Optimization of Large-Scale QoS-Aware Service Compositions

Florian Rosenberg[*] Max Benjamin Müller[†] Philipp Leitner[†] Anton Michlmayr[†] Athman Bouguettaya[*] Schahram Dustdar[†]

[*]CSIRO ICT Centre
GPO Box 664, Canberra, 2601 ACT, Australia
firstname.lastname@csiro.au

[†]Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1, 1040 Vienna, Austria
lastname@infosys.tuwien.ac.at

***Abstract*—We present an optimization approach for service compositions in large-scale service-oriented systems that are subject to Quality of Service (QoS) constraints. In particular, we leverage a composition model that allows a flexible specification of QoS constraints by using constraint hierarchies. We propose an extensible metaheuristic framework for optimizing such compositions. It provides coherent implementation of common metaheuristic functionalities, such as the objective function, improved mutation or neighbor generation. We implement three metaheuristic algorithms that leverage these improved operations. The experiments show the efficiency of these implementations and the improved convergence behavior compared to purely randomized metaheuristic operators.**

I. Introduction

Service-oriented systems have gained momentum as a means to implement robust and interoperable distributed applications [1]. Enterprises often adopt the Service-Oriented Architecture (SOA) paradigm to implement inter-organizational and mission critical business processes [2], [3], [4]. Such processes typically compose a large number of atomic services into so-called composite services. For example, consider a large online electronic reseller, called PCSell, who is assembling built-to-order PCs and selling a wide range of electronic equipment. If it uses a service-based infrastructure, it would typically consume services from business partners. These include payment services (such as Paypal), Customer Relationship Management (CRM) services (such as Salesforce) or shipping services. PCSell's business success is highly dependent on a reliable IT system that must satisfy rigid non-functional properties such as 24/7 availability, efficient execution time or high throughput. Non-functional properties are specified as QoS constraints on composite services. QoS includes both technical measures (e.g., response time of individual services or availability [5]) and business-related attributes (e.g., order fulfillment time). While some of these constraints are required, other constraints are considered "nice to have", i.e., desirable but not critical. The specification and enforcement of QoS constraints on a composite service level requires the necessary means to optimize such compositions. Additionally, it requires a pool of alternative services with different QoS or services offered by different business partners (e.g., shipping services).

We argue that an efficient runtime optimization of large-scale QoS-aware compositions is of particular importance since a number of QoS attribute values are dynamic. These would typically depend on environmental factors. For example, response time depends on input data, server load, and network latency. Therefore, service compositions need to be continuously monitored and re-optimized in case the overall QoS is not satisfactory. It is important that runtime re-optimization is triggered whenever QoS changes are reported by monitoring modules [5], [6]. This calls for efficient optimization methods. Additionally, the optimal solution for the composition problem is not usually required. Instead, it is important that all QoS constraints are fulfilled and solutions be generated efficiently.

Most existing optimization approaches for QoS-aware service compositions use some form of global optimization to find the best solution in terms of QoS [7], [8], [9], [10]. While this is definitely useful for small compositions, it incurs a significant performance penalty if applied to larger compositions, especially for runtime optimization. Contrary to global optimization, a metaheuristic combines basic heuristic methods in higher level frameworks to efficiently and effectively exploring a search space [11]. There is ample evidence regarding the applicability of metaheuristics for large-scale optimization problems [11], [12]. However, no coherent approach is available to demonstrate the effectiveness and efficiency of metaheuristics for large-scale QoS-aware service composition.

We propose a comprehensive framework for optimizing large-scale QoS-aware compositions at runtime. It uses an extensible QoS model and a flexible specification of QoS constraints by leveraging constraint hierarchies, a mechanism that categorizes the importance of constraints using expressive labels [13]. We combine flexible QoS constraint specification with efficient metaheuristic optimization algorithms in a novel way. In particular, we propose a coherent way to evaluate service candidates and solutions, i.e., an assignment of concrete services to abstract services in a composition. In addition, we present efficient heuristics for modifying existing solutions. They further improve the quality of generated solutions compared to purely randomized ones. Thanks to their generic nature, the proposed heuristics can be used across a variety of existing algorithms such as Genetic Algorithms (GA), Simulated Annealing (SA) and Tabu Search (TS). The experiments demonstrate the efficiency and improved convergence behavior compared to purely randomized implementations.

978-0-7695-4126-6/10 $26.00 © 2010 IEEE
DOI 10.1109/SCC.2010.58

IEEE computer society

The rest of the paper is structured as follows: Section II presents the composition model for this approach. Section III describes our metaheuristic framework. Section IV evaluates the approach. Section V describes the related work. Section VI provides concluding remarks and highlights some future work.

II. QoS-Aware Composition Model

In this section, we present the composition model upon which the foundation for the metaheuristic optimization approach is built. We briefly describe our approach for specifying QoS constraints for a composite service.

A. Composition Model

A composite service CS consists of a set of n abstract services $S_j = \{s_1, s_2, \ldots, s_n\}$. An *abstract service* is a non-executable service describing the core functionality in terms of operations, input, output, pre- and postconditions [14]. These abstract services are composed using various control flow structures such as sequences, loops, conditional or parallel execution. This specification is called an *abstract composite service*. For each abstract service s_j, a set of m concrete services $C_{ij} = \{c_{1j}, c_{2j}, \ldots, c_{mj}\}$ is available that implements the corresponding abstract service. To enact an abstract composite service, a selection and binding of each abstract service to a concrete service in the composition needs to be established. This task is commonly known as service selection.

B. QoS Model

Service selection can be improved by considering the candidate services' QoS for an abstract service s_j. Our approach provides an extensible QoS model, thus we use a vector Q to denote all available QoS attributes in the system. Q^k then refers to the k-th QoS attribute within this vector. Each candidate has a vector of QoS values, where q_{ij}^k denotes the QoS value of C_{ij} (attribute type Q^k). In this work, we focus on two categories: (1) operational QoS attributes, e.g., response time, throughput, availability or security; (2) business-related attributes such as price, reputation or order fulfillment time.

We generally distinguish between three different types of QoS attributes (also known as their *dimension*) as explained in Table I. The functions (column 2) return $true$ whenever a given attribute has this dimension. They are particularly important for the optimization phase (see next section) to determine what attribute value is better than another one. It also ensures extensibility of the QoS model because the optimization framework can determine the dimension of each QoS attribute at runtime.

C. QoS Constraints

We propose a flexible model for incorporating QoS constraints into the optimization process. Constraints usually apply to different services or the whole composition. They can also have different importance ranging from "nice to have" to required constraints. Constraints applied to specific regions of a composite service are not the scope of this paper.

Dimension	Function	Description	Examples
metric ascending	$asc(Q^k)$	a higher value is better	availability, throughput
metric descending	$desc(Q^k)$	a lower value is better	response time, price
nominal	$nom(Q^k)$	no natural ordering	security protocol

TABLE I
QoS Dimensions

Basically, two different constraint types can be specified on the abstract composite service level [15]. *Global constraints* express QoS constraints for the overall abstract composite service. *Local constraints* are applied only to single services within the abstract composition. We leverage constraint hierarchies [13] to capture the flexibility of specifying constraints. A constraint hierarchy H is a multiset of labeled constraints. H_0 denotes the *required constraints* in H (also called *hard constraints*). The sets H_1,…,H_n are defined for the hierarchy levels 1…n representing weaker, so-called *soft constraints* labeled with different strengths. Each level expresses constraints that are equally important. We use hierarchy levels with the following strengths: $\{required, strong, medium, weak\}$. However, this can be adapted to represent different levels.

We specify constraints as follows: A global constraint (GC) for a given QoS attribute Q^k can be written as a tuple $GC^k = \langle gc\_rv^k, gc\_s^k \rangle$. The first element represents the desired value of the constraint, e.g., for a response time constraint $q_{rt} \leq$ 1500msec, gc_rv^k is 1500. The second element represents the strength of the constraint, e.g., $required$ if the QoS constraint needs to be satisfied. Local constraints (LCs) are specified in the same way for a given abstract service S_i and QoS attribute Q^k: $LC_i^k = \langle lc\_rv_i^k, lc\_s_i^k \rangle$.

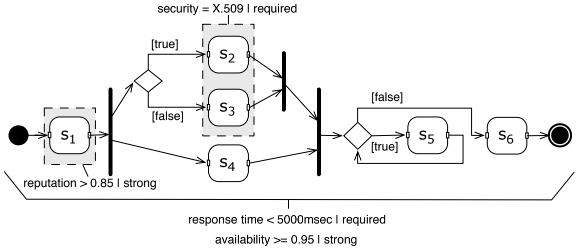

security = X.509 | required
[true]
s2
s3
[false]
s1
s4
[false]
s5
[true]
s6
reputation > 0.85 | strong
response time < 5000msec | required
availability >= 0.95 | strong

Fig. 1. Abstract Composition Example

Figure 1 shows a simple composition example as an Unified Modeling Language (UML) activity diagram with six abstract services denoted as s_1 to s_6 and a number of constraints. Abstract service s_1 defines a LC desiring a high reputation which can be useful for example when selecting online reseller services. Abstract services s_2 and s_3 define a LC for security, e.g., when selecting credit card services. The GCs address the response time and availability to ensure adequate performance.

III. Metaheuristic-based Optimization

Based on the composition and QoS model, we introduce the basic concepts to evaluate service candidates and solutions. We define the objective function and introduce solution modification as a common concept shared across metaheuristic algorithms. Finally, we show how Simulated Annealing (SA) leverages these concepts for a specialized implementation.

A. Overview

A metaheuristic is an iterative generation process which guides a subordinate heuristic by combining different concepts for exploring and exploiting the search space. They are an effective method for finding near-optimal solutions for large-scale problems [16]. Different metaheuristic algorithms have been developed over the years, such as Genetic Algorithms (GA), Simulated Annealing (SA), Tabu Search (TS) and Ant Colony Optimization (ACO). Each metaheuristic proposes a different strategy, however, some basic principles and commonalities are shared. The goal of the optimization is to find a solution that maximizes or minimizes a user-specified objective function. This function is used as a black-box to evaluate whether a given solution is satisfied. Some metaheuristics maintain one solution at a time (such as SA) while others use multiple (such as GA and TS). Specific heuristics, called *generators*, are used to generate initial solutions. During the optimization, new solutions are usually generated through *mutation*. If a solution is based on a current one, it is called a *neighbor*. Both are usually implemented using probabilistic methods. For example, GAs generate multiple solutions and use another heuristic to select which solutions will be combined by using a so-called *crossover* operator. All metaheuristics generally keep track of the current optimum and can be terminated based on different termination conditions (e.g., time or number of iterations). SA is discussed in detail in Section III-D.

We leverage these commonalities to build a common framework for representing a solution, modeling the objective function and modifying solutions. In this paper, we focus specifically on two aspects: First, a solution needs to be represented in combination with the definition of an objective function. We incorporate QoS constraints using penalty values if constraints are violated. Second, a neighbor generation function needs to be specified to generate new and hopefully better solutions based on a given solution by applying small modifications. We refer to this requirement as *solution modification*.

B. Candidate and Solution Evaluation

Before describing how solutions are generated and improved, we describe how to evaluate service candidates and solutions. The goal is to generate aggregated values for both, the candidate and the solution that combines QoS and penalty values.

1) Candidate Evaluation: Each service candidate needs a score and penalty value as general quality indicators for the optimization process. Different quality attributes have different scales, therefore, they need to be normalized. We use Simple Additive Weighting (SAW), similar to [7]. The normalized value nq_{ij}^k of q_{ij}^k is computed so that the best normalized values are equal to 0, the worst equal to 1. Thus, higher normalized values indicate worse quality. Nominal attributes are exceptional, since the candidate either has the same value as the required one or not. Additionally, each QoS attribute can have a weight to express its importance in the global context. For example, availability could be given a higher weight than reputation because the latter values are specified by the provider and might therefore not be fully trusted. The user-defined attribute weights w^k are used for computing the weighted normalized value as follows: $wnq_{ij}^k = nq_{ij}^k \cdot w^k$. All these weights can be predefined as constant values during the optimization and are usually application dependent.

Local constraint violations are incorporated by calculating a penalty for a candidate. Therefore, we map the strength levels of the constraint hierarchy H to numeric values. We use the following mapping: $\{required = 20, strong = 10, medium = 5, weak = 2.5\}$. For simplicity, we assume that $lc\_s_i^k$ and also gc_s^k return the mapped value of a local and global constraint, respectively. The penalty value pv_{ij}^k for q_{ij}^k is set to the $lc\_s_i^k$ of the local constraint if the candidates' QoS value is less (asc) or greater (desc) than the local required value $lc\_rv_i^k$. For nominal attributes, the situation is more complex. Only in case each candidate has the same nominal value, the aggregated value will be equal to this value, otherwise it will be $undefined$. Thus, if a global constraint for a nominal attribute exists, all candidates selected for each abstract service must "support" the required value gc_rv^k, otherwise the constraint cannot be satisfied. Thus, a global constraint of a nominal attribute has the same semantic as applying a local constraint for the same nominal attribute to each abstract service with the same required value. It should be clear, that a user must not define contradictory global and local nominal constraints. Therefore, the penalty value for nominal attributes is the sum of both local $lc\_s_i^k$ and global constraint strengths gc_s^k in case they are violated. In any other case, the penalty is set to 0. These penalty values need to be combined into a single value reflecting the overall local penalty of a candidate C_{ij}: $\mathrm{clp}(C_{ij}) = \sum_{k=1}^{d} pv_{ij}^k$. The overall *candidate score* of C_{ij} can then be computed as follows: $\mathrm{csc}(C_{ij}) = \sum_{k=1}^{d} wnq_{ij}^k + \mathrm{clp}(C_{ij})$. It consists of the weighted normalized values and the overall local penalty. The first part is helpful to find candidates satisfying local constraints. The second part reflects the overall quality of the candidate from a local and global perspective.

2) Solution Evaluation: Solutions created during the optimization process also require a score and a penalty. The score should reflect the overall QoS quality and the global/local constraint satisfaction. Additionally, the overall global and local penalty values should reflect the constraint violations. These values allow the optimization to distinguish between good and bad solutions. This leads the search process into promising directions. The evaluation of the objective function is, therefore, highly critical for the success and quality of the optimization.

First of all, the aggregated QoS value aq_s^k of solution S has to be computed for each quality attribute Q^k. This is achieved by providing aggregation functions specifically for each QoS attribute and composition construct. For example, abstract services s_1 and s_3 in Figure 1 are conditionally executed,

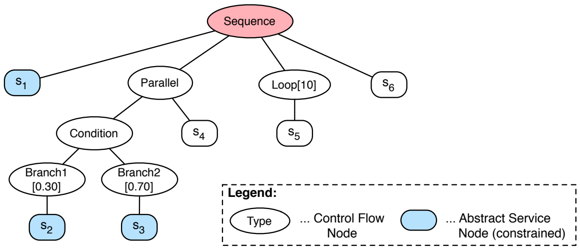


Fig. 2. Normalized Abstract Composition for Figure 1

therefore, it is necessary to know the execution probabilities of both branches. Another example is the loop enclosing t_5, where the expected loop iterations need to be specified. Therefore, we represent the composition as a tree (Figure 2) by getting rid of the cycle caused by the loop. We call this a *normalized abstract composition* as shown in Figure 2 for the composite service depicted in Figure 1. Internal tree nodes represent the composition constructs (either sequential, parallel, conditional or loop execution) whereas leaf nodes represent the abstract services. The aggregation algorithm traverses the tree and calculates the aggregated QoS values for each attribute based on the atomic aggregation functions. Due to space reasons, we cannot discuss the aggregation formulas and algorithm and refer to [15] for a more detailed discussion.

Based on the aggregated value, we also need to normalize the aggregated QoS values because of the different scales. The normalized values naq_s^k of aq_s^k are calculated using SAW similar to the candidate normalization. Thus, the best normalized values are equal to 0, the worst equal to 1. Again, nominal attributes do not need to be normalized, these attributes are only relevant for the penalty values. Similar to the candidate selection, the user-specific attribute weights w^k are used to compute a weighted normalized value for each Q^k as follows: $wnaq_s^k = naq_s^k \cdot w^k$. These values are then combined into a single *solution quality value* (sqv) as follows: $\mathrm{sqv}_s = \sum_{k=1}^{d} wnaq_s^k$.

As the next step, the penalty values for violating global constraints have to be computed. It should reflect both, the distance to the required values gc_rv^k and the constraint strengths gc_s^k. For nominal attributes, let Inv_s be a set containing all (invalid) candidates selected where $q^k_{i\,\mathrm{sel}(S,i)} \neq gc\_rv^k$ for all abstract services S_i. Let spv_s^k denote the penalty value for the aggregated value aq_s^k:

$$\mathrm{spv}_s^k = \begin{cases} \dfrac{gc\_rv^k}{aq_s^k} \cdot gc\_s^k & \text{if } \mathrm{asc}(Q^k) \text{ and } aq_s^k < gc\_rv^k \\ \dfrac{aq_s^k}{gc\_rv^k} \cdot gc\_s^k & \text{if } \mathrm{desc}(Q^k) \text{ and } aq_s^k > gc\_rv^k \\ \dfrac{|Inv_s|}{n} \cdot gc\_s^k & \text{if } \mathrm{nom}(Q^k) \text{ and } aq_s^k \neq gc\_rv^k \\ 0 & \text{otherwise} \end{cases}$$

Since there are potentially different scales for different metric attributes, the difference between the required and actual value cannot be directly used. Therefore, ratios are used to express how far a value is away from satisfying a constraint. For ascending metric attributes, the global required value gc_rv^k is divided by aq^k, for descending metric attributes the fraction is inversed. These values are always greater than 1 and become higher the worse the found values get (in a scale-independent way due to the fraction). For nominal attributes the distance to satisfaction is computed by dividing the number of selected invalid candidates by the number of abstract services n. By multiplying these values with the constraint strength gc_s^k, we receive an expressive penalty value.

Now the global penalty values need to be combined into a single value reflecting the *solution global penalty*. It sums up the penalty values for each attribute Q^k: $\mathrm{sgp}(S) = \sum_{k=1}^{d} spv_s^k$. Since we do not deal with global constraints only, we also have to consider local penalties. Therefore, the overall local penalty for solution S is the sum of the local penalties of all selected candidates (expressed as $C_{i\,\mathrm{sel}(S,i)}$): $\mathrm{slp}(S) = \sum_{i=1}^{n} \mathrm{clp}(C_{i\,\mathrm{sel}(S,i)})$.

Solutions that violate constraints must always have a higher score than those satisfying all, thus the following *solution penalty* value is computed:

$$\mathrm{sp}(S) = \begin{cases} \sum_{k=1}^{d} w^k & \text{if}(\mathrm{sgp}(S) + \mathrm{slp}(S)) > 0 \\ 0 & \text{otherwise} \end{cases}$$

In case there is any constraint violation, the value is equal to the sum of the weights of all defined QoS attributes. By adding this value to the overall score (see below), a solution that violates any constraint will always have a higher score than one that does not.

Finally, the different factors need to be combined into a single expressive value. The overall solution score that represents the objective function is then computed as follows:

$$\mathrm{ssc}(S) = w\_sqv \cdot sqv_s + w\_sgp \cdot \mathrm{sgp}(S) + w\_slp \cdot \mathrm{slp}(S) + \mathrm{sp}(S)$$

The score is higher the "worse" the solution is (with 0 being the lower bound). The user-defined weights w_sqv, w_sgp and w_slp again allow to customize and balance the different factors and can be defined as constants.

C. Solution Modification

The convergence of purely random solution modification operators is really slow, particularly for large composite services. Situations were only very specific abstract service/candidate assignments will lead to constraint satisfaction are extremely improbable to be fulfilled in a specific amount of time. Although random modification might lead to near-optimal results in the end, it needs much time and computational resources. Therefore, we propose an approach to better meet the requirements of QoS-aware service composition. The approach still uses randomization to a certain degree to avoid ending up in local optima.

1) Improved Mutation: The main task of mutation and neighbor generation is the creation of a new solution based on an existing one by modifying it in a way that it is more likely to fulfill all QoS constraints. Algorithm 2 shows the basic outline of our improved mutation algorithm. For a given *solution*, it checks whether it satisfies all global and local constraints (line 2). If satisfied, the mutation is trying to improve the overall quality (see Algorithm 2). Otherwise, it checks whether the solution's global or local penalty is higher (line 5) and then either improves the global or local penalty (line 6 and line 8). In the following, we discuss all operations in more detail.

Algorithm 1 Improved Mutation Algorithm

```
 1: procedure MUTATE$_{imp}$($solution$)
 2:     if sat($solution$) then
 3:         IMPROVE_QUALITY($solution$)
 4:     else
 5:         if sgp($solution$) $>$ slp($solution$) then
 6:             IMPROVE_GLOBAL_PENALTY($solution$)
 7:         else
 8:             IMPROVE_LOCAL_PENALTY($solution$)
 9:         end if
10:     end if
11: end procedure
```

a) Quality Improvement: Algorithm 2 depicts the Improve_Quality operation that tries to improve the overall quality of a solution which is known to satisfy all constraints.

Algorithm 2 Quality Improvement

```
1: procedure IMPROVE_QUALITY($solution$)
2:     $tot\_ex$ = GET_EXCHANGE( )
3:     $per\_attr$ = split $tot\_ex$ proportionally between attributes
4:     for all $\langle attr, ex\rangle$ in $per\_attr$ do
5:         METRIC_ATTRIBUTE_IMPROVEMENT($solution$,$attr$,$ex$)
6:     end for
7: end procedure
```

First, the algorithm determines the number of abstract service/candidate assignment exchanges in total by selecting a random number between 1 and a configurable parameter `maxExchanges`. In line 3 the total number of exchanges is split, as the quality of multiple attributes should be improved at the same time. The number of exchanges per attribute is based on the proportional selection mode using the following value: $w^k \cdot naq^k_{solution}$. Thus, attributes with a high weight (important) and a high normalized aggregated value (i.e., far away from the best possible value) are selected with higher probability. The choice to include normalized values is motivated by the fact that QoS attributes with a very low normalized value can be hardly improved. Afterwards the Metric_Attribute_Improvement operation is used for each $\langle attr,\ ex\rangle$ pair. It is responsible for improving the solutions quality of exactly one metric attribute. The basic idea is to exchange assigned candidates with low (ascending) or high (descending) QoS values by better ones. Nominal QoS attributes cannot be further improved after constraint satisfaction because normalized aggregated values of nominal attributes are always equal to 0, there is no chance they are selected. Therefore, they are not considered for quality improvement. However, they need to be considered in case an solution does not satisfy all the constraints as discussed next.

b) Improving the Global Penalty: The implementation of the Improve_Global_Penalty operation is similar to the one above. Instead of improving the overall quality, it tries to lower the global penalty, thus nominal attributes have to be considered. It basically needs to determine the unsatisfied global constraints and then split the number of exchanges proportionally among them based on the the weight of the global constraint gc_w. This ensures that constraints of higher importance are more probable to be selected and are, therefore, earlier satisfied in the optimization process. Finally, for each exchange pair, the referenced QoS attribute is improved using the Metric_Attribute_Improvement or Nominal_Attribute_Improvement operation.

c) Improving the Local Penalty: If the solution's penalty is lower or equals the overall solution's local penalty, the mutation's local penalty needs to be improved (cf. Algorithm 1, line 5). This is achieved by running a tournament selection [12] to determine the abstract service/candidate assignments that need to be exchanged. Generally, for the selection of m items out of n, m tournaments of a given size are held. In each tournament the best item wins and is added to the result set. This requires a *better* function to be specified that returns true if the randomly selected item is better than the current best item. In the Improve_Local_Penalty operation, the exchange is performed if the randomly selected candidate assigned to an abstract service has a higher local penalty (clp) as the currently best candidate in the tournament. For each identified abstract service where the candidate needs to be exchanged, the algorithms checks if other candidates with a lower penalty exist. If this is the case, another tournament is executed among those candidates to select the candidate with the lowest penalty.

2) Neighbor Generation: In contrast to mutation, neighbor generation creates multiple copies of an existing solution (based on a configurable *amount* parameter). Other than that, it is similar to the improved mutation because it leverages the same set of operations as mutation (algorithm not shown for space reasons). In fact, it could be implemented by invoking the mutation for each clone. Unfortunately, all neighbors would then fall into the same improvement category (quality, global and local penalty). However, this is not optimal if both global and local constraints are violated. Therefore, we use a proportional selection to distribute whether the local or the global penalties should be improved (proportional to the solution's penalty value). This broadens the neighborhood search and leads to better results in case of TS where multiple neighbors are generated in each iteration.

D. Specialized Metaheuristic Algorithm

We have created two implementation of each metaheuristic algorithm. The basic version uses purely random operators whereas the specialized version uses the aforementioned operations. Due to space limitations, we only show the implementation of SA. The implementation of the improved TS closely follows its basic version. The GA is more complex because an efficient crossover operator needs to be provided but this is out of scope of this paper.

SA originates from the process of physical annealing of solids [17]. A crystalline solid is heated and then cooled down very slowly. This allows the solid to achieve its most regular possible crystal lattice configuration having its minimum lattice energy state. During cooling, different states with different energies are passed through. Typically, transitions to states with lower energy are preferred, still it is possible that a transition to a state with a slightly higher energy is taken. The Metropolis criterion is used for simulating this behavior [18]. Thus, the objective function is minimized with the help of a fictitious temperature. The temperature is in this case a controllable parameter of the algorithm. The implementation is illustrated in Algorithm 3.

Algorithm 3 Simulated Annealing

```
 1: function SA_Optimize($composition$, $condition$)
 2:     $solution$ = Create_Initial_Solution($composition$)
 3:     $temp$ = Get_Initial_Temperature($composition$)
 4:     Evaluate($solution$)
 5:     $best$ = $solution$
 6:     while $condition$ does not fire do
 7:         $neighbour$ = Generate_Neighbor($solution$)
 8:         Evaluate($neighbour$)
 9:         if Accept($neighbour$) then $solution$ = $neighbour$
10:             if $solution$ better than $best$ then
11:                 $best$ = $solution$
12:             end if
13:         end if
14:         Update($temp$)
15:     end while
16:     return $best$
17: end function
```

First, an initial solution and temperature are determined (lines 2–3). Instead of using a random initial solution (as common for basic SA implementations), we use a *best candidates* initial solution. This means that the candidate with the best score is selected for each abstract service. Thus, for abstract service S_i the candidate with the lowest score is chosen because a lower score is better (m_i is the number of candidates for S_i):

$$\min \operatorname{csc}(C_{ij}) \quad \forall j \leq m_i$$

The candidate's score is a good indicator for its value for the whole composite service. This usually leads to a feasible solution much faster than the random one.

The Get_Initial_Temperature operation follows the geometric cooling schedule [12]. It is computed as the difference between the maximum and minimum solution score. For determining the maximum score the following estimation is used. A solution S_{max} is created such that for each abstract service S_i the candidate with the worst score is selected:

$$\max \operatorname{csc}(C_{ij}) \quad \forall j \leq m_i$$

The score of S_{max} is then used as maximum score. As the theoretically lowest score is 0, this value is used as an estimation for the minimum score. Therefore the initial temperature $temp$ becomes: $temp = \operatorname{ssc}(S_{max})$.

The solution is then evaluated and stored as $best$ solution (lines 4-5). As long as the termination condition (line 6) is not satisfied, the improvement of the solution is performed. In each iteration, a neighbor solution $neighbour$ is generated (line 7). For creating a neighbor of the current solution, we do not rely on randomized neighbor generation as the standard SA implementation. Instead, the improvement SA neighbor generation leverages the $\texttt{MUTATE}_{imp}\,(neighbour)$ as described in Algorithm 1.

The newly created solution is accepted as base solution for the next iteration either if it is better than the current solution or if the Metropolis criterion is satisfied (lines 8–9). If $neighbour$ is even better than the $best$ one, $best$ is accordingly updated (lines 10–12). Following the typical acceptance criterion of simulated annealing the current $solution$ is replaced by $neighbour$ in the following cases:

- the $neighbour$'s score is better than the $solution$'s one:
$$\operatorname{ssc}(neighbour) < \operatorname{ssc}(solution)$$
- the metropolis criterion is satisfied. This is implemented by generating a random double number in (0,1) and checking whether the following statement is true:
$$rand \leq \exp\left(\frac{\operatorname{ssc}(solution) - \operatorname{ssc}(neighbour)}{temp}\right)$$

At the end of each iteration, the temperature is updated (line 14) by decreasing it in each iteration by using the following function (again following the geometric cooling schedule): $temp = temp \cdot \alpha, \quad \alpha \in [0, 1)$. If the termination condition becomes satisfied, the $best$ solution so far is finally returned (line 16).

IV. Implementation and Evaluation

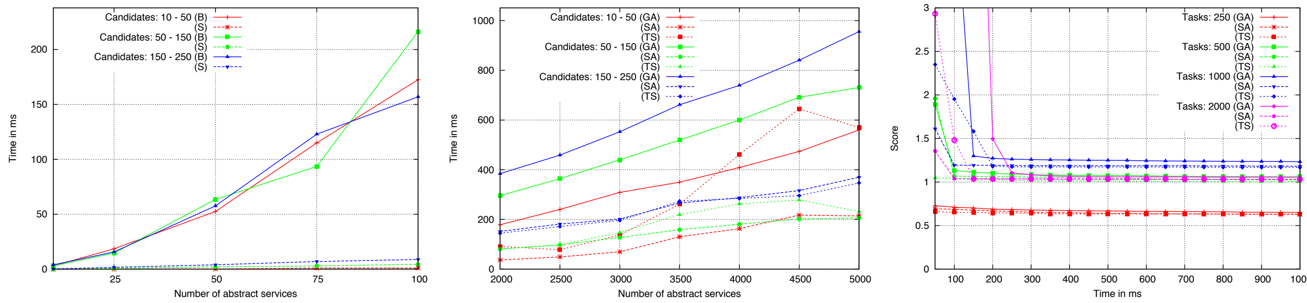
We implemented the optimization framework using C#.NET by focusing on extensibility with respect to the composition constructs, termination conditions, the QoS model and its aggregation functions. Additionally, we provided connectors to the VRESCo platform [19]. VRESCo offers a simple domain-specific language called VCL (Vienna Composition Language) that implements the composition model described in this paper [15]. By providing these connectors, the optimization framework can work with compositions specified in VCL, read the QoS data and perform the optimization.

Based on the implementation and integration in VRESCo, we performed a number of experiments by using generated compositions with different complexity. All metaheuristic implementations (GA, SA and TS) use a set of "optimal" configuration parameters. They were determined by experimenting with different parameter settings to find the lower and upper bound. Due to space reasons, a detailed description of this pre-evaluation is out of scope.

All experiments were executed on a Intel Core 2 Duo E6600 CPU with 2.4 GHz and 2 GB DDR II 667 MHz RAM using Windows XP SP3 as operation system and .NET 3.5 SP1.

A. Composition Tree Generation

A composition generator generates large in-memory VRESCo composition trees by using the following distribution of composition constructs: Sequence - 0.50, conditional - 0.30, parallel - 0.15, loop - 0.05. The maximum branches for sequence nodes are set to 10, for parallel nodes to 5 and for conditionals to 5. The maximum number of rounds of a



(a) Basic vs. Specialized SA

(b) Performance for 2500-5000 Abstract Services

(c) Overall Score Comparison

Fig. 3. Performance and Score Comparison

loop node is set to 5. A recursive algorithm then generates the desired compositions of the given size. Additionally, we use the following QoS attributes (values in brackets denote the distribution): response time [100-300], price [10-20], availability [0.95-1], accuracy [0.95-1], throughput [100-150], reliable messaging [0,1] and security [0,1,2,3,4]. The algorithm randomly selects 5 QoS attributes as global constraints. Local constraints are added to abstract services with a probability of 0.30. The number of constraints per abstract service is determined by selecting a random value between 1 and 5. The required QoS values are determined by randomly selecting a quality of 50% to 70% of the attribute range for each abstract service. For nominal attributes, always the highest nominal value is chosen.

B. Performance Comparison

Figure 3(a) shows the difference between the basic SA implementation (B) and the specialized implementation (S) using the efficient operators developed in this work. The specialized version is almost up to a 100 times faster. Higher numbers of abstract services are not shown because the runtime of the basic SA grows fast and it would not be readable in one plot.

Figure 3(b) depicts the average time for finding a feasible solution for compositions ranging from 2500 to 5000 abstract services (smaller ones are omitted for readability). For each abstract service size 200 random compositions are built and optimized. If no solution could be found within 5000 ms, the optimization process was stopped. It can be seen that the GA performs worse than the SA and TS approaches. For small candidate sizes, SA is even a bit faster than TS. However, the SA approach was not able to find a feasible solution for compositions consisting of 1250 abstract services twice. The TS approach also did not find a feasible solution for two generated compositions consisting of 2000 abstract services. In contrast, the GA always identified a solution satisfying all constraints. Interestingly, the standard deviation of SA and TS is much higher for small candidate sizes (not shown as a plot). A possible explanation is that the scores computed for the candidates during the candidate evaluation are much more expressive if a lot of candidates are available for each abstract service. Thus, the initial solution built is less vulnerable to superficially "good" candidates that eventually prove to be unsuitable for finding a solution satisfying all constraints.

Finally, the score depending on the time passed is evaluated. Typical abstract service sizes have been chosen: 250, 500, 1000, 2000. For each size, 20 compositions were created and optimized for 1000 ms each. Figure 3(c) shows the average scores reached by each algorithm for candidate sizes of 10-50 per abstract service. For larger candidate sizes comparable results are reached, thus, they are not further shown and discussed. As can be easily determined, the GA needs the most time for converging, while SA generally shows the fastest convergence behavior. TS can be found in between. After about 300 ms, the score improved only very little for all metaheuristics.

V. Related Work

Most QoS-aware composition approaches focus on variants of linear programming methods to solve the optimization problem. Little work on metaheuristic approaches is available that design operators specifically for the optimization of large-scale QoS-aware compositions.

Zeng et al. [7] present two approaches, one focusing on the local optimization, the second one on global optimization. For solving the global optimization problem, they use Integer Linear Programming (ILP). The objective function is formulated to maximize the overall QoS and global QoS constraints are considered. Since not all QoS aggregation formulas are linear, they need to be linearized (e.g., by using the natural logarithm). This can have an (unwanted) impact on the objective function, although the constraints still hold. Berbner et al. [20] implemented three heuristics: (i) a MILP formulation solved through IP relaxation; (ii) one that randomly replaces abstract service/candidate assignments; (iii) an SA algorithm to temporarily accept worse solutions to leave local optima and possible find the global optimum. Ardagna and Pernici [9] leverage Mixed Integer Linear Programming (MILP) to propose an improved version of the work from Zeng et al. They introduce advanced concepts such as loop peeling and negotiation mechanisms to address situation where no feasible solution can be found. Alrifai and Risse [10] propose a solution where they decompose global QoS constraints into local constraints with conservative upper and lower bounds. These local constraints are resolved by using an efficient distributed local selection strategy.

All of the aforementioned approaches do not leverage metaheuristics but they provide a good performance for small and a reasonable performance for medium scale compositions. However, they do not support the definition of user-defined hard and soft QoS constraints.

Canfora et al. [21] were the first to use GA for the optimization of QoS-aware compositions. The main advantage with respect to ILP approaches is that any kind of constraint and objective/fitness function (not only linear ones) can be used as in our approach. The results show that their GA implementation scales much better than the ILP approach present for example by Zeng et al. Jaeger and Muehl [22] present another GA-based approach that uses a slightly different fitness function and a special crossover variant where the fitter parent is dominant. Their results show reasonable performance but the solutions are far from being optimal. Kobti and Zhiyang [23] present a GA and Cultural Algorithm (CA). While the first one is similar to Canfora et al., the latter uses a global belief space and an influence function that should accelerate the convergence of the population. However, they do not consider global nor local constraints.

In contrast to our work, existing metaheuristics based approaches generally do not provide operators specialized on the optimization problem of large-scale QoS-aware compositions. Instead, standard operators such as two-points-crossover for GA are used, which solely depend on randomness. While this might be applicable to find a good solution for small-sized problems, it does not lead to satisfying results when applied to larger-scale problems. Convergence will also suffer, since the chance of creating a better solution by randomly replacing abstract service/candidate assignments is limited.

VI. Conclusions

In this paper we have proposed a metaheuristic framework for the QoS-aware composition problem. It is novel in the sense that it supports (i) a flexible way to specify the QoS requirements using constraint hierarchies; (ii) a extensible and user-defined QoS model and (iii) improved mutation and neighbor generation heuristics. We have implemented three specialized versions of well-known metaheuristics, namely GA, SA and TS. The optimization outperforms existing approaches with regard to the optimization time, especially for large-scale services. This makes the framework useful for runtime composition and re-composition in enterprise environments because the overhead is minimal for small to medium size compositions. Additionally, it is applicable in domains where performance and large-scale systems are dominant, such as scientific computing or computational science.

Our future work will explore the problem of dependent and conflicting QoS attributes and how this can be considered during the optimization. Additionally, we will extend the support for nominal QoS attributes. Currently, QoS constraints on nominal attributes require all services in a composition to support them. However, this is very inflexible and leads to frequent constraint violations.

References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges," *IEEE Computer*, vol. 40, no. 11, pp. 38–45, 2007.

[2] R. A. Paul, "DoD Towards Software Services," in *Proc. of the 10th IEEE Intl Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'05)*. IEEE Computer Society, 2005, pp. 3–6.

[3] I.-L. Yen, H. Ma, F. B. Bastani, and H. Mei, "QoS-Reconfigurable Web Services and Compositions for High-Assurance Systems," *Computer*, vol. 41, no. 8, pp. 48–55, 2008.

[4] C. Huemer, P. Liegl, R. Schuster, H. Werthner, and M. Zapletal, "Inter-organizational systems: From business values over business processes to deployment," in *2nd IEEE Intl. Conf. on Digital Ecosystems and Technologies (DEST'08)*, 2008, pp. 294 –299.

[5] F. Rosenberg, C. Platzer, and S. Dustdar, "Bootstrapping Performance and Dependability Attributes of Web Services," in *Proc. of the IEEE Intl. Conf. on Web Services (ICWS'06)*. IEEE Computer Society, Sep. 2006, pp. 35–44.

[6] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "Comprehensive QoS Monitoring of Web Services and Event-Based SLA Violation Detection," in *Proc. of the 4th Intl. Workshop on Middleware for Service Oriented Computing (MW4SOC'09)*. ACM Press, Nov. 2009.

[7] L. Zeng, B. Benatallah, A. H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "QoS-Aware Middleware for Web Services Composition," *IEEE Trans. Softw. Eng.*, vol. 30, no. 5, pp. 311–327, 2004.

[8] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient algorithms for Web services selection with end-to-end QoS constraints," *ACM Trans. Web*, vol. 1, no. 1, p. 6, 2007.

[9] D. Ardagna and B. Pernici, "Adaptive Service Composition in Flexible Processes," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 369–384, 2007.

[10] M. Alrifai and T. Risse, "Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition," in *Proc. of the 18th Intl. World Wide Web Conf. (WWW'09)*. ACM, 2009, pp. 881–890.

[11] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, 2003.

[12] J. Dréo, P. Siarry, A. Pétrowski, and E. Taillard, *Metaheuristics for Hard Optimization*. Springer, 2006.

[13] A. Borning, B. Freeman-Benson, and M. Wilson, "Constraint hierarchies," *Lisp and Symbolic Computation*, vol. 5, no. 3, pp. 223–270, 1992.

[14] B. Medjahed and A. Bouguettaya, "A dynamic foundational architecture for semantic web services," *Distributed and Parallel Databases*, vol. 17, no. 2, pp. 179–206, 2005.

[15] F. Rosenberg, P. Celikovic, A. Michlmayr, P. Leitner, and S. Dustdar, "An End-to-End Approach for QoS-Aware Service Composition," in *Proc. of the 13th IEEE Intl. EDOC Conference*, 2009.

[16] I. H. Osman and G. Laporte, "Metaheuristics: A bibliography," *Annals of Operations Research*, vol. 63, pp. 513–623, Oct. 1996.

[17] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671–680, 1983.

[18] T. F. Gonzalez, Ed., *Handbook of Approximation Algorithms and Metaheuristics*. Chapman & Hall/CRC, 2007.

[19] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "End-to-End Support for QoS-Aware Service Selection, Binding and Mediation in VRESCo," *IEEE Transactions on Services Computing*, 2010, (to appear).

[20] R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz, "Heuristics for QoS-aware Web Service Composition," in *Proc. of the IEEE Intl. Conf. on Web Services (ICWS'06)*. IEEE Computer Society, 2006, pp. 72–82.

[21] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, "An Approach for QoS-aware Service Composition based on Genetic Algorithms," in *Proc. of the Intl. Conf. on Genetic and Evolutionary Computation (GECCO'05)*. ACM, 2005, pp. 1069–1075.

[22] M. C. Jaeger and G. Mühl, "QoS-based Selection of Services: The Implementation of a Genetic Algorithm," in *Proc. of the KiVS 2007 Workshop: Service-Oriented Architectures und Service-Oriented Computing (SOA/SOC)*, 2007, pp. 359–370.

[23] Z. Kobti and W. Zhiyang, "An Adaptive Approach for QoS-Aware Web Service Composition Using Cultural Algorithms," in *Proc. of Advances in Artificial Intelligence (AI 2007)*. Springer, 2007, pp. 140–149.