The RADO Approach to Quality-Driven Service Composition—Approximating the Pareto-Frontier in Polynomial Time

Immanuel Trummer
Artificial Intelligence Laboratory
Ecole Polytechnique Fédérale de Lausanne
immanuel.trummer@epfl.ch

Boi Faltings
Artificial Intelligence Laboratory
Ecole Polytechnique Fédérale de Lausanne
boi.faltings@epfl.ch

ABSTRACT

In Quality-Driven Service Composition, tasks from an abstract workflow are assigned to concrete services such that workflow QoS are optimized. The following three properties are desirable for a corresponding algorithm. First, the run time is ideally bounded by a polynomial in the number of workflow tasks and service candidates. Second, the algorithm provides formal guarantees on how close the returned solution is to the optimal one. Third, the algorithm is able to return a representative set of Pareto-optimal solutions instead of only one. The user can choose among them or apply arbitrary filter or sort operations in successive steps. We present the first algorithm that has all three features. We analyze its formal properties and evaluate the algorithm experimentally. We find that our algorithm is competitive from the theoretical as well as from the practical perspective.

1. INTRODUCTION

Large scale, public registries for Web services have emerged in recent years. Examples include the Biocatalogue\(^1\) SSE\(^2\) and Seekda\(^3\) which currently advertises over 28,000 Web services. With the growing number of available services, the chances to find several functionally equivalent services for a given task increase. Non-functional quality of service (QoS) parameters can be considered to choose between them. This leads to Quality-Driven Service Composition (QDSC)\(^4\). QDSC starts from an abstract workflow. Every abstract task is associated with a set of services that differ only in their non-functional properties such as response time and reliability. Selecting one service for every task makes the abstract workflow executable. The properties of the selected services will determine the non-functional properties of the executable workflow. The goal of QDSC is to select services such that the QoS of the executable workflow are optimized.

We present a novel algorithm for QDSC named RADO (we resolve this acronym at the end of Section 4). Our algorithm distinguishes itself from existing approaches since it combines the three desirable formal properties which we outline now. First, QDSC takes place at run time, so efficiency is crucial. Therefore, the run time of a QDSC algorithm should be bounded by a polynomial in the number of workflow tasks and service candidates. Many heuristic approaches to QDSC have this feature (e.g. \(^5\)). However, they fail to provide formal guarantees on how close the returned result is to the optimum. Formal guarantees on approximation quality are the second desirable property. Ideally, users should in addition be able to tune approximation quality, trading run time for optimality. Approaches like the one by Zeng et al. \(^11\) find the optimal solution. However, they fail to guarantee polynomial run time (since QDSC is NP-hard \(^10\)). Finally, QDSC corresponds to a multi-dimensional optimization problem (different QoS properties of the resulting executable workflow). Most existing approaches combine those quality dimensions into one utility value using user-defined weights. They return only the solution with (approximately) best utility value. However, it may be difficult for users to choose those weights in advance, requiring several runs before the user is satisfied. Therefore, another desirable feature for a QDSC algorithm is to return a representative set of Pareto-optimal solutions instead of only one (e.g. \(^6\)).

The original scientific contributions of this paper are i) a novel algorithm for QDSC that combines the three desirable formal properties outlined before, ii) a thorough theoretical analysis of the algorithm, proving the bounds for run time and approximation quality, and iii) a detailed experimental evaluation of our approach. We introduce our formal model in Section 2. Then we introduce the basic algorithm in Section 3 and formulate several requirements on sub-procedures of this algorithm. In Section 4 we discuss how to satisfy these requirements and present a corresponding algorithm. We analyze the formal properties of our algorithm in Section 5. We evaluate our algorithm experimentally in Section 6 and finally compare with related work in Section 7.

2. SYSTEM MODEL AND ASSUMPTIONS

We make several fundamental assumptions that are common in QDSC (e.g. \(^11\)). First, we assume that the service registry supports semantic matchmaking between tasks and

\(^1\)http://www.biocatalogue.org/
\(^2\)http://services.eoportal.org/
\(^3\)http://webservices.seekda.com/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Copyright 20XX ACM X-XXXXX-XX-XX/XX ...$10.00.
services. Second, we assume that reliable information about non-functional properties of services is available. Third, we assume that estimates concerning the probability of different workflow execution paths is available. Such information is essential for QDSC as outlined for instance by Ardagna et al. [2]. It can be either estimated in domain-specific ways or from the traces of past executions. We will present a formal model for QDSC which is based on these assumptions.

2.1 Workflows and services
We denote by $\mathcal{S}$ the set of Web services in the registry. Workflows aggregate services and define the control flow between service invocations. QDSC starts with an abstract workflow, meaning that service invocations are not yet bound to concrete services. We describe an abstract workflow as a binary tree $(V, E)$ $(V$ are the nodes and $E$ the edges with $E \subseteq V \times V)$. We explain in Section 5.1 how to transform non-binary workflow trees into binary ones and Figure 2 illustrates the process. Every node in the tree represents an activity. We call the root node the root activity, if $(p, c) \in E$ then we call $p$ the parent activity of $c$ and $c$ the child-activity of $p$. The leaf nodes represent simple activities (also called tasks), they symbolize one service invocation. We define a boolean function $isSimple(v)$ for $v \in V$ which yields true if and only if $v$ is a leaf node. As the workflow is abstract, every simple activity $v \in V$ is associated with a set of functionally equivalent services that we denote by $candidates(v) \subseteq \mathcal{S}$. We call the inner nodes of the tree complex activities, they define the control flow between their child activities. Every complex activity $v \in V$ is associated with a control flow construct $construct(v) \in \{SEQ, PAR, CHC\}$. The semantic of $construct(v) = SEQ$ is that the child activities should be executed sequentially, $PAR$ symbolizes parallel execution and $CHC$ conditional execution of exactly one of the child activities. We do not consider loop constructs as loops can be either unrolled [11] or peeled [3]. For the remainder of the paper we assume a fixed abstract workflow. A binding $b$ is a function $b : \{v \in V : isSimple(v)\} \rightarrow \mathcal{S}$ that maps simple activities to concrete services such that for every simple activity $v : b(v) \in candidates(v)$. An abstract workflow together with a binding can be executed. Note finally, that our model would not be detailed enough for executing the workflow since we do for instance not represent choice conditions.

Example 1. Figure 1 shows an example workflow tree. Our workflow is based on the Personalized TV Guide example by Yu et al. [9]. The workflow consists of 3 simple and 4 complex activities. It first retrieves cinema and TV program in parallel by two service invocations. The retrieved program items form the input to a recommender service (we assume that information about movies the user liked in the past is available). Depending on whether the recommended item is a TV series or a movie, one of two services is invoked to obtain additional information.

2.2 Quality of service
We assume a fixed set of QoS attributes $A$. We represent QoS values as positive real numbers. QoS values for all attributes can be represented as vectors within the quality space $\mathbb{Q} = \mathbb{R}^{|A|}_+$. For $\bar{q} \in \mathbb{Q}$, we denote by $\bar{q}_a$ the component of the vector that refers to attribute $a \in A$. The QoS properties of services are given (e.g. advertised in the registry). For an abstract workflow together with a binding, we can estimate the QoS of the whole workflow. The QoS properties of complex activities are aggregated from the QoS of the child activities. The aggregation function depends on the attribute and the construct of the complex activity. By $AgQ(v)$ we denote the aggregation function for every complex activity $v \in V$. We denote the aggregation function for one specific attribute $a$ for $v$ by $AgQ_a(v)$. We consider maximum, minimum, weighted sum and product as aggregation functions. The weights for the weighted sum must be chosen out of the interval $[0,1]$. The weights may for example depend on the transition probabilities (which we do not represent explicitly in our model) if the complex activity is of type choice. Now we can define the recursive function $\breve{Q}$ which estimates the QoS of every activity $v \in V$ for a binding $b$:

$$\breve{Q} : (v, b) \rightarrow \begin{cases} QoS \text{ for } b(v) \in \mathcal{S} & \text{if isSimple(v)}, \\
AgQ(v)(\breve{Q}(v1), \breve{Q}(v1)) & \text{if } (v, v1), (v, v2) \in E.
\end{cases}$$

(1)

Different QoS attributes have different value domains. In addition, some of them are positive attributes (meaning: a higher value corresponds to better QoS, e.g. reliability) while others are negative attributes (meaning: a higher value corresponds to worse QoS, e.g. cost). We will scale QoS values to the interval $[0,1]$ such that $1$ always represents the best QoS. The scaling function $\sigma$ maps from the QoS space $\mathbb{Q}$ to the scaled quality space $\bar{Q} = [0,1]^{|A|}$. We scale with regards to a multi-dimensional scaling range which is defined by two QoS vectors, the lower bound $\breve{l} \in \mathbb{Q}$ and the upper bound $\breve{u}$. We define $\sigma$ component-wise for every attribute $a$:

$$\sigma_a : (\breve{q}_a, \breve{l}_a, \breve{u}_a) \rightarrow \begin{cases} [\bar{q}_a, \bar{u}_a] & \text{if a positive,} \\
[\bar{l}_a, \bar{q}_a] & \text{if a negative.}
\end{cases}$$

(2)

Example 2. Figure 2 shows how we scale response time to lower ($\breve{l}$) and upper ($\breve{u}$) bound. Note that a higher value in $\bar{Q}$ is scaled to a lower value in $\bar{Q}$ since response time is
The goal in PQDSC is to find sets of bindings (instead of single bindings) that approximate the Pareto-frontier. The input to PQDSC is the target precision \( \varepsilon \) and a tuple describing an abstract workflow, QoS aggregation for the activities and the available services:

\[
I = (V, E, \text{isSimple}, \text{candidates}, \text{construct}, A g Q, S)
\] (4)

A solution to the PQDSC problem is a set of bindings \( B \) such that i) no binding \( b_1 \) in \( B \) is dominated by another binding \( b_2 \in B \) (\( b_1 \not\subseteq b_2 \)), ii) the Pareto error of \( B \) for the root node \( v_r \in V \) is bounded by \( \varepsilon t \). The Pareto error is always defined with regards to a specific scaling range. We define the total quality range \( [q_l, q_u] \) for a workflow such that the QoS at \( v_r \) of any possible binding \( b \) is included in this range: \( \forall a \in A : q_l a \leq \bar{Q}(v_r, b)_a \leq q_u a \). We distinguish QoS attributes with bounded and unbounded domain. For attributes with bounded domain, there are lower and upper bounds on the (non-scaled) possible QoS which are independent from the workflow and the available services. An example is reliability whose values are always between 0 and 1. An example for an unbounded attribute is response time, since it can grow to any value given the right service candidates and workflow. For bounded attributes we set \( q_l a \) to the lower and \( q_u a \) to the upper bound of their a-priori value domain (e.g. for reliability \( q_l a = 0, q_u a = 1 \)). Denote by \( B \) the set of all possible bindings for the workflow. For unbounded attributes, we set the lower bound of the total quality range to the minimum QoS value over all bindings: \( q_l a = \min_{b \in B} \bar{Q}(v_r, b)_a \). We set the upper bound correspondingly: \( q_u a = \max_{b \in B} \bar{Q}(v_r, b)_a \). Note that those values can be calculated efficiently as shown by Zeng et al. We use \( q_l \) and \( q_u \) as defined here for the remainder of the paper. Now we can formulate requirement ii) on the result set \( B \) formally: \( PEr(B, v_r, q_l, q_u) \leq \varepsilon t \).

### 2.5 Parameters describing problem size

We characterize the size of the problem input by the following parameters (that we use during asymptotic run time analysis etc.). \( N = |V| \) is the number of activities in the abstract workflow, \( S = \sum_{v \in V} |\text{candidates}(v)| \) (assuming \( \text{candidates}(v) = 0 \) if \( \text{isSimple}(v) = \text{false} \)) the number of service candidates. For clarification: \( N \) denotes the number of activities in a binary workflow tree. However, transforming a non-binary workflow tree into a binary one doubles at most the number of activities. Therefore, it does not matter for the asymptotic analysis whether we consider the number of activities in the original or in the binary tree. By \( A = |A g Q| \) we denote the number of attributes and by \( ct \) again the target precision. For our complexity analysis, we consider \( N, S \) and \( \varepsilon^{-1} \) as variables and \( A \) as constant. New Web services may be added to the registry at any time, the number of workflow activities and the target precision are chosen by the user. Introducing new QoS attributes (that are not calculated from existing ones) is more difficult. The monitoring infrastructure must be adapted to measure the new QoS attribute and data about services must be collected (even if service providers advertise the QoS themselves, some verification mechanism should be implemented). Benchmarks in QDSC typically use low numbers of QoS attributes in comparison to the number of services and tasks (e.g. 5 attributes, up to 80 tasks and up to 40 services per task).
3. BASIC ALGORITHM

In this section, we present the main procedure of our algorithm for solving the PQDSC problem. We will informally introduce a recursive algorithm that finds all possible bindings (with their QoS) for a given workflow. We refine this algorithm step-by-step to guarantee polynomial run time and bounded Pareto error. At the end of this section, we present the pseudocode of the final algorithm. Note that our algorithm relies on a sub-procedure for filtering bindings on which we formulate several formal requirement. We will introduce this sub-procedure in the next section.

3.1 Finding all possible bindings

We construct bindings recursively: bindings for complex activities are constructed by combining bindings of their child activities. Figure 5(a) shows how we treat simple and complex activities.

(a) Construct all possible bindings

- Simple Activity
  - Service Candidates: \( S \)
  - \( |B| \leq P \)
  - \( |B| = |S| \)

- Complex Activity
  - \( B = B_{C1} \cup B_{C2} \)
  - \( |B| \leq P^2 \)

(b) Guarantee polynomial run time

- \( \text{PEr}(B, v) \leq \varepsilon \)
- \( \text{PEr}(B, p) \leq \varepsilon_1 + \varepsilon_2 + \varepsilon \)
- \( \text{PEr}(B, v) = 0 \)
- \( \text{PEr}(B, p) \leq \varepsilon_1 + \varepsilon_2 \)

(c) Guarantee bounded Pareto error

- \( \text{PEr}(B_{C1}, c_1) \leq \varepsilon_1 \)
- \( \text{PEr}(B_{C2}, c_2) \leq \varepsilon_2 \)

Figure 5: Refinement of recursive procedure

3.2 Guaranteeing polynomial run time

We need to filter out bindings in order to achieve polynomial time. In this section, we formulate several requirements for a filtering function which filters out bindings for every activity (for every tree node). We will show that the overall algorithm has polynomial run time and can guarantee bounds on the Pareto error (in Section 3.3) if the filtering function complies with these requirements. We will present ideas how to implement this function in Section 4.

Requirement 1. The time complexity for the filtering must be polynomial in \( N, S, \) and \( \varepsilon t^{-1} \) (under the assumption that the same holds for the size of the input set for the filtering).

Requirement 2. The size of the filtered set is restricted (independently from the size of the input set!) by a polynomial \( P = P(N, S, \varepsilon t^{-1}) \).

Figure 5(b) illustrates how the size of the result set develops if the latter requirement is satisfied. For every activity, the algorithm returns at most \( P(N, S, \varepsilon t^{-1}) \) bindings. The joined set can be constructed in \( O(P^3) \), the time complexity of the filtering is polynomial, too, and \( N \) activities need to be treated. Therefore, the total time complexity is polynomial in \( N, S, \) and \( \varepsilon t^{-1} \), too. We analyze time and space complexity of our algorithm in Section 5 in more detail.

3.3 Guaranteeing approximation quality

Filtering guarantees polynomial run time. However, removing a binding from a set may increase the Pareto error of this set. Additionally—since we construct bindings recursively—removing one binding may increase the Pareto error of the set which is constructed for the parent activity. We are ultimately interested in the Pareto error at the root activity, scaled to the total quality range at the root. However, we must filter sets of bindings for other activities as well. The question is, which range we should scale to in order to compare bindings for other activities than the root while filtering them. We therefore introduce the critical range.

Definition 2. (Critical Range) The critical range is defined for every activity \( v \in V \) by two QoS vectors, the critical lower bound \( \bar{c}(v) \in Q \) and the critical upper bound \( \bar{c}(v) \in Q \) such that for the root activity \( vr \in V \) \( \bar{c}(vr) = q_l \) and \( \bar{c}(vr) = q_u \) and the following requirement holds.

Requirement 3. Let \( B_{c1} \) a set of bindings for \( c1 \in V \) and \( B2 \) a set of bindings for \( c2 \in V \) and \( p \) the common parent of \( c1 \) and \( c2 \). Set \( \varepsilon p = P \text{Er}(B1 \cap B2, p, \bar{c}(p), \bar{c}(p)) \), the error for the joined set, \( \varepsilon 1 = P \text{Er}(B1, c1, c1), \bar{c}(c1)), \) and 
\( \varepsilon 2 = P \text{Er}(B2, c2, c2), \bar{c}(c2)) \). Then \( \varepsilon p \leq \varepsilon 1 + \varepsilon 2 \) (5)

The latter is a requirement on the definition of critical ranges. We show how to compute them accordingly in Section 4. Our last requirement concerns the filtering function again.

Requirement 4. The filtering function can be tuned by a parameter \( \varepsilon \) which is an upper bound on the added Pareto error during filtering. Let \( B \) a set of bindings for activity
Algorithm 1 RADO for Pareto QDSC

1: // I = (V,E,isSimple,candidates,construct,AgQ,S)
2: function RADO(I, εt,v)
3:   res ← ∅ // Represents result set
4:   // Test: simple or complex activity?
5:   if isSimple(v) then
6:     // For all candidate services
7:     for all s ∈ candidates(v) do
8:       resItem ← {(v, s), ⃗Q(s)}
9:     res ← res ∪ {resItem}
10:   end for
11: else
12:   activity is complex
13:   \{(c1, c2) ∈ \{c ∈ V | (v, c) ∈ E\} | c ∈ S\}
14:   res1 ← RADO(I, εt, c1)
15:   res2 ← RADO(I, εt, c2)
16:   res ← join(res1, res2, AgQ(v))
17:   end if
18: return filter(res, v, εt/|V|)
19: end function

Algorithm 2 Join two result sets

1: function JOIN(res1, res2, ⃗F)
2: res ← ∅
3: for all (b1, q1) ∈ res1 do
4:   for all (b2, q2) ∈ res2 do
5:     b ← b1 ∪ b2 // Union of bindings
6:     q ← ⃗F(q1, q2) // Aggregate QoS
7:     res ← res ∪ {{b, q}}
8: end for
9: end for
10: return res
11: end function

Figure 6: Function calls for example workflow

4. FILTERING BINDINGS

In this section, we show how to define critical ranges and how to filter bindings so that the 4 requirements from the last section are satisfied. If we mention scaled QoS values in the following, we always mean scaled to the critical range (we show how to construct the critical range in Section 4.1). Our filtering is based on a discretization of scaled QoS vectors.

Definition 3. (Discretization) Discretization designates a transformation from the scaled QoS space SQ to a finite, discrete space DQ, the discretized quality space. Discretization depends on a parameter ε. A higher ε means lower precision during discretization and DQ = \{0, \ldots, \lfloor 1/ε \rfloor \}.

We define the scaling function \( \tilde{\delta} \) : SQ → DQ for \( \tilde{\delta} \in SQ \):

\[
\tilde{\delta}(\epsilon) : SQ \rightarrow DQ \quad \text{for} \quad \tilde{\delta} \in SQ:
\]

We can translate our definition of dominance between bindings to the discretized space. Denote by \( dq1, dq2 \in DQ \) the discretized (with precision \( \epsilon \)) scaled QoS of bindings \( b1 \) and \( b2 \) for activity \( v \in V \) (both scaled to \( \tilde{\delta}(v) \), \( \tilde{\delta}(v) \)). We write \( b1 \sqsupseteq b2 \) if \( \forall a \in A : dq1_a \geq dq2_a \) and \( \exists a \in A : dq1_a > dq2_a \).

While filtering a set of bindings \( B \), we remove a binding \( b1 \in B \) if there exists a binding \( b2 \in B \) such that \( b2 \sqsupseteq b1 \). If we have two bindings with the same discretized QoS, then we non-deterministically choose one of them to remove. During filtering, we might increase the Pareto error of the set. However, for every binding \( b1 \) that we remove from \( B \), we know that another binding \( b2 \) will ultimately remain in the filtered set and \( b2 \) is worse at most by \( \epsilon \) than \( b1 \) in every QoS dimension. Therefore, the Pareto error may at most increase by \( \epsilon \).

Example 4. Figure 6 shows the calls to the join and filter functions that are issued for our example workflow. We do not show the calls to the filter function for simple activities to improve readability. The stylized Pareto-frontiers represent sets of bindings with their QoS.
4.1 Constructing the critical range

We must define critical ranges such that Requirement 3 is satisfied. The critical range for the root activity is equivalent to the total quality range. For all other activities, we define the critical range of the critical range of the parent activity and the total quality range. Figure 7 illustrates the process: for every attribute we calculate total quality ranges for all activities in a bottom-up traversal (as described by Zeng et al. [11]). Then we calculate critical ranges for every activity in a top-down traversal.

We must calculate the critical range differently for different types of attributes. Table 1 shows how we classify attributes into 4 categories (type 1 to type 4). We distinguish attributes with regards to value domain and the set of aggregation functions they use (over all possible constructs). Let $$c \in V$$ one activity for complex activity $$p \in V$$. Fix an attribute $$a \in A$$. For the remainder of this Subsection we omit attribute index and vector arrows and implicitly refer to this attribute (e.g. $$cl(c) = cl_a(c)$$). If $$a$$ is aggregated as weighted sum in $$p$$, we denote by $$W$$ the weight for $$c$$. Table 2 shows the formulas for calculating critical ranges. Those depend on the attribute type of $$a$$ (“Attr. Type” in Table 2) and the aggregation function for $$a$$ in $$p$$ (“Agg. Fct.”). Note that for attributes with a bounded domain, the critical range is the same for all activities. Also note that for attributes of type 3, the lower bound of the critical range always corresponds to the lower bound of the total quality range. For attributes of type 4, the upper bound of the critical range always corresponds to the upper bound of the total range. We will prove in Section 4.2 that this definition of critical ranges guarantees that Requirement 3 is satisfied.

4.2 Pseudocode

The filtering function (Algorithm 3) implements the filtering as described before in an efficient manner. We assume that critical ranges have been calculated before and are accessible as $$cl(v)$$ and $$cu(v)$$ (for $$v \in V$$). The input consists of the set of items to filter (one item is a binding with its QoS), the activity to which the bindings refer, and the precision parameter $$\varepsilon$$. Note that we call the filtering function with $$\varepsilon = ct/N$$ as explained in Section 3.3. The function uses the two array variables $$dominated$$ and $$itemsTable$$ where indices corresponds to QoS vectors in the discretized quality space $$DQ$$. Array $$dominated$$ saves a boolean value for every QoS vector indicating whether a binding was already inserted that dominates this QoS field. The array $$itemsTable$$ saves at most one binding for every QoS index. The bindings in $$toFilter$$ are treated one after the other. A binding is inserted into $$itemsTable$$ if its discretized QoS values are not dominated by a binding inserted before (for $$\delta$$ see Def. 3). If a new binding is inserted, the dominated bindings must be deleted in $$itemsTable$$ and the corresponding QoS vectors in $$dominated$$ must be marked as dominated. This accomplishes Algorithm 4. The propagation function takes the two arrays $$dominated$$ and $$itemsTable$$ as input (call-by-reference), as well as the discretized QoS vector of the new binding. It is recursive and treats fields which are dominated by the new entry. The propagation stops once the functions reaches either the boundaries of the array (one of the indices is smaller than zero) or fields which have already been marked.

Example 5. Figure 8 illustrates the filtering. It shows the
5. FORMAL ANALYSIS OF ALGORITHM

Algorithm 3 Filter out dominated and equivalent bindings

1: function FILTER(toFilter, v, ε)
2:    // Scaled, discretized QoS - index space for arrays
3:    DQ ← {0, ..., |δ/v|}^A
4:    // Initialise arrays
5:    for all i ∈ DQ do
6:        dominated[i] ← false
7:        itemsTable[i] ← ∅
8:    end for
9:    // Filter bindings
10:   for all (b, q) ∈ toFilter do
11:       s_i(q) ← s_i(q; c_i(v), c_i(u)) // scale to critical range
12:       i ← s_i(q) // discretize
13:       if dominated[i] = false then
14:           propagate(dominated, itemsTable, i)
15:            itemsTable[i] ← (b, q)
16:        end if
17:   end for
18:   // Retrieve non-dominated items from table
19:   for all i ∈ DQ do
20:       res ← res ∪ {itemsTable[i]}
21:   end for
22:   return res
23: end function

Algorithm 4 Mark fields in QoS space as dominated

1: function PROPAGATE(dominated, itemsTable, i)
2:    itemsTable[i] ← ∅
3:    if dominated[i] = false then
4:        dominated[i] ← true
5:        for all a ∈ A do
6:            j ← i
7:            j_a ← j_a - 1
8:            if j_a ≥ 0 then
9:                propagate(dominated, itemsTable, j)
10:           end if
11:        end for
12:    end if
13: end function

We summarize the preparatory steps to perform. i) First, we must transform the workflow tree into an equivalent binary tree. This can be accomplished by two top-down traversals in the tree. In the first traversal, we delete complex activities with only one child activity. In the second traversal, we expand activities with more than two children into a chain of newly added complex activities with two children each. Figure 2 shows a simple example. While changing the tree, we must eventually adapt weights for QoS aggregation such that every binding will have equivalent QoS at the root for original and binary tree. ii) We must calculate the total quality ranges for every activity. Zeng et al. [11] describe how to do that efficiently. iii) We must calculate the critical ranges for every activity. This can be done by one top-down traversal of the tree using the formulas from Section 4.1. After the preparation, Algorithm 3 is executed (input parameter v is the root activity for the first call). Algorithm 4 uses Algorithm 3 and Algorithm 5 as auxiliary functions. Algorithm 6 uses Algorithm 4 as sub-procedure.

5.2 Approximation precision

We fix tree arbitrary activities c_1, c_2, p ∈ V for the remainder of this section such that p is the parent activity of c_1 and c_2. We examine how to construct the critical range for c_1 from the critical range of p. We simplify the notation. Note first of all that critical ranges are constructed independently for different attributes. The equations in the remainder of this section refer either to one attribute or must hold for all attributes. We omit attribute indices and vector signs and will clarify in the accompanying text for which attributes a formula holds. By F we denote the aggregation functions for the parent F = AgQ[p], by σ_p the scaling function for the parent σ_p(q) = σ(q; c_p(v), c_p(u)), by σ_1 the scaling function for c_1, σ_1(q) = σ(q; c_1(v), c_1(u)), and by σ_2 the scaling function for c_2 in the analogical manner. The Pareto error is always calculated with respect to the critical range of the corresponding activity, we therefore omit the boundary parameters: PErr(B, v) = PErr(B, v, c_i(v), c_i(u)). The following theorem provides a reformulation of Requirement 4 that we use for constructing critical ranges.

**THEOREM 1.** If (9) is satisfied for all attributes ∀a ∈ A, all possible QoS values q_1, q_2 ∈ R, and for all ε ∈ R with 0 ≤ ε ≤ qu(c_1) - q_1, then the propagation of the Pareto error is bounded and (9) holds.

\[
|σ_p(F(q_1 + ε, q_2)) - σ_p(F(q_1, q_2))| \leq |σ_1(q_1 + ε) - σ_1(q_1)|
\]

(8)

\[
PErr(B_1 ⊗ B_2, p) \leq PErr(B_1, c_1) + PErr(B_2, c_2)
\]

(9)

**Proof.** Set εp = PErr(B_1 ⊗ B_2, p), ε1 = PErr(B_1, c_1) and ε2 = PErr(B_2, c_2). Let b be a Pareto-optimal binding for p. Assume that b is not Pareto-optimal for the child activities c_1 and c_2 (or both). Then we can always find a binding bo with equivalent QoS in p (Q(p, b) = Q(p, bo)) and bo is Pareto-optimal for c_1 and c_2 (improving the QoS of a child activity can only improve the QoS at the parent). Let bo = b_1 ⊗ b_2 such that b_1 is a Pareto-optimal binding for c_1 and b_2 Pareto-optimal for c_2. Let b_1 ∈ B_1 a binding for c_1 whose scaled QoS value is not worse by more than ε1 in every QoS dimension compared with b_1 (such a binding must exist since PErr(B_1, c_1) = ε1). Let b_2 ∈ B_2 a binding for c_2 whose scaled QoS value is not worse by more than ε2 in every QoS dimension compared with b_2 (such a binding must exist since PErr(B_2, c_2) = ε2). We have now introduced the complete algorithm and can resolve the acronym. RADO abbreviates Recursive Assembly of Discretized (Pareto) Optima and therefore captures the core ideas of our approach.

5. FORMAL ANALYSIS OF ALGORITHM

We analyze the precision of the algorithm (Section 5.2), the time (Section 5.3) and space complexity (Section 5.4). We start with a summary of the RADO algorithm and the necessary preparatory steps.

5.1 Summary of algorithm
dimension comparing with \( b_2 \). Now imagine that we change the binding for \( p \) in two steps from \( b_1 \cup b_2 \) to \( b_1 \cup b_2 \): first, we replace \( b_1 \) by \( b_1 \), then we replace \( b_2 \) by \( b_2 \). Fix an arbitrary attribute \( a \in A \) and denote \( q_1 = Q(c_1, b_1) \) and \( q_2 = Q(c_2, b_2) \). Further, let
\[
\Delta 1 = \sigma_p(F(q_1, q_2)) - \sigma_p(F(q_1, q_2)) \quad (10)
\]
\[
\Delta 2 = \sigma_p(F(q_1, q_2)) - \sigma_p(F(q_1, q_2)) \quad (11)
\]
We have
\[
\sigma_p(F(q_1, q_2)) - \sigma_p(F(q_1, q_2)) = \Delta 1 + \Delta 2 \quad (12)
\]
Now if \( \tilde{q}_1 \leq q_1 \) then \( \Delta 1 \leq 0 \leq \varepsilon 1 \). If \( \tilde{q}_1 > q_1 \) then \( \Delta 1 = |\Delta 1| \) and applying \( (8) \) yields \( |\Delta 1| \leq |\sigma(q_1) - \sigma(q_1)| \) and again \( \Delta 1 \leq \varepsilon 1 \). By a similar reasoning, we find that \( \Delta 2 \leq \varepsilon 2 \). Therefore, \( (9) \) holds.

We show that defining critical ranges as in Table 2 guarantees that \( (8) \) is satisfied.

**Theorem 2.** Assume that for an attribute with bounded domain (type 1 or 2) the critical ranges are defined according to Table 4 (\( \forall(v) = cu(v) = cu \) and \( \forall(v) = cl(v) = cl \) for all activities \( v \in V \)). Then this implies \( (8) \).

**Proof.** In the following we treat the case of positive attributes. The proof for negative attributes is analogue. We can simplify the definition of the scaling function \( \sigma \) for our specific choice of critical ranges. First, note that QoS values are always contained within the critical range. Therefore, \( |\sigma(v) - \sigma(v)| \) is constant over all activities \( v \in V \). Hence
\[
\sigma_1(q) = \sigma_2(q) = \sigma(q) = \frac{q - cl}{cu - cl} \quad (13)
\]
So \( \sigma \) is a linear transformation and \( (14) \) and \( (15) \) hold.
\[
\sigma(F(q_1 + \varepsilon, q_2)) - \sigma(F(q_1, q_2)) = \sigma(F(q_1 + \varepsilon, q_2) - F(q_1, q_2)) \quad (14)
\]
\[
\sigma(q_1 + \varepsilon) - \sigma(q_1) = \sigma(\varepsilon) \quad (15)
\]
\( \sigma \) is monotone (for positive attributes), so we just have to show \( (10) \) which implies \( (8) \). According to
\[
\sigma(F(q_1 + \varepsilon, q_2) - F(q_1, q_2)) \leq \varepsilon \quad (16)
\]
Equation \( (10) \) is obviously satisfied for minimum, maximum, and the weighted sum (with weights between 0 and 1). It is satisfied for the product, because in this case the attribute is of type 1, so \( cl = 0, cu = 1 \) and \( q_2 \leq 1 \) which implies \( (17) \).
\[
F(q_1 + \varepsilon, q_2) - F(q_1, q_2) = \varepsilon \cdot q_2 \leq \varepsilon \quad (17)
\]

**Theorem 3.** Assume an attribute with unbounded domain (type 3 or 4) and the critical ranges are defined according to Table 3. Then this implies \( (8) \).

**Proof.** We assume that the attribute is positive and of type 4. Equation \( (9) \) compares \( \sigma_p(q_1) \) and \( \sigma_p(F(q_1, q_2)) \). Figure 9 shows the development of both functions in \( q_1 \) for a fixed (but unknown) \( q_2 \). Assume now that \( F \) is the maximum between \( q_1 \) and \( q_2 \). This case is represented in Figure 12a. The curve for \( \sigma_1 \) (lower half of Figure 12a) follows directly from the definition of \( \sigma_1 \) and because we assume

![Figure 9: Propagation of QoS changes](image)
type 4. The cases of negative attributes and of attributes of type 3 can be proven analogously. □

5.3 Time complexity
For the following analysis we assume that elementary arithmetic, list and tree operations can be performed in constant time. We already made sure that the time complexity is polynomial in $N$, $S$ and $et^{-1}$ by the design of our algorithm. Now we want to find out the exact polynomial. All preparations (transformation to binary tree, calculating total and critical quality ranges) can be done in linear time in $N$ and $S$ (and do not depend on $et^{-1}$). Algorithm \( \text{Algorithm 1} \) is executed once for every activity, $N$ times. A simple activity can be treated in $O(S)$ (since $S$ is the maximum number of service candidates and one binding per service is added). For complex activities, the result sets from two recursive calls are joined and then filtered. The size of one result set is restricted by $(N/et)^{A_{-1}}$ ($A = 1$ as exponent since for every fixed assignment for $A = 1$ QoS attributes, there can be at most one non-dominated item in the result). Two result items can be combined in $O(1)$ (since $A$ is a constant), Algorithm \( \text{Algorithm 2} \) is therefore in $O(N(1/et)^{2A-2})$. For complex activities, Algorithm \( \text{Algorithm 3} \) is called on the result of the join operation. Checking one binding (whether it should be inserted) and inserting it in \text{itemsTable} is in $O(1)$. However, in the worst case all cases in array \text{dominated} have to be marked by Algorithm \( \text{Algorithm 4} \). There are $(N/et)^{A}$ cases. In total, the filtering for one complex activity is performed in $O(N(1/et)^{2A-2} + (N/et)^{A})$. The filtering for a simple activity is performed in $O(S + (N/et)^{A})$. The total time complexity of RADO is

$$O(N \cdot (S + (N/et)^{A} + (N/et)^{2A-2})).$$

(22)

5.4 Space complexity
For this analysis, we assume that elementary data types such as numbers and booleans are in $O(1)$ space. The space requirements for the preparatory phases are linear in $N$ and $S$. Algorithm \( \text{Algorithm 1} \) is executed once for every activity. It first constructs a set of bindings. For simple activities, the constructed set of bindings is in $(S)$ space. For complex activities, two sets of filtered bindings have to be stored (res1 and res2) which are in $O((N/et)^{A_{-1}})$ space. Additionally, a joined set of bindings is constructed which is in $O((N/et)^{2A-2} + S)$ space. We now explain how to change the algorithm in order to reduce the space requirements. We did not integrate this optimization into the pseudocode in Section 4.4 in order to improve the readability. We integrate Algorithm \( \text{Algorithm 2} \) and Algorithm \( \text{Algorithm 3} \) directly into Algorithm \( \text{Algorithm 1} \). Instead of first constructing all possible bindings and filtering this set afterwards, we filter every binding directly after its construction. Therefore, we only need to save sets with at most $(N/et)^{A_{-1}}$ bindings for every instance. Note that this optimization improves the space complexity but does not change time complexity. In addition, we have to save the arrays \text{dominated} and \text{itemsTable} for every activity. Note that bindings for simple activities can be represented by a service ID, therefore in $O(1)$ space. Bindings for complex activities are always assembled out of two bindings (since the workflow tree is binary), one for each child activity. They can be represented by pointers to those two bindings which is also in $O(1)$ space (so at the root node, every binding is represented by a pointer-tree). In summary, the space-optimized version of our algorithm is in

$$O(N \cdot (S + (N/et)^{A}))$$

(23)

6. EXPERIMENTAL EVALUATION
We evaluate our algorithm for QDSC and PQDSC. We are not interested in worst-case guarantees and assume the filtering function is called with a fixed $\varepsilon$ (instead of $et/N$). We name RADO instances accordingly (e.g. RADO 0.5 if $\varepsilon = 0.5$). All benchmarks were executed on a 2.53 GHz Intel Core Duo processor with 2.5 GB RAM running Windows 7.

In QDSC, the goal is to find a binding that maximizes the utility function and respects the constraints. The utility function is a weighted sum over the scaled QoS, constraints correspond to lower bounds on the scaled QoS values. We compare RADO with the genetic algorithm (GA) by Canfora et al. \[5\]. We use the same Java library and the same parameters except that we vary the number of generations (GA $x$ for GA with $x$ generations). We generate test cases randomly (workflow, constraints, utility weights, and a register). We considered the QoS attributes time, availability, and throughput. We used the QWS dataset \[8\] for generating registry instances. The QoS vectors from the dataset are randomly assigned to functional categories. Figure 10 reports arithmetic mean average values over 50 test cases (50 for every number of workflow activities) with 10% confidence intervals. Figure 10(a) shows the average processing time in milliseconds, Figure 10(b) the percentage of returned solutions that satisfy all constraints, and Figure 10(c) reports the utility value of the returned binding (algorithms return the binding with highest utility among the discovered bindings that satisfy the maximum number of constraints). The tendencies are the same for all algorithms. Higher numbers of activities increase the processing time and diminish the chances to find good solutions. In all categories the RADO instances perform significantly better than the GAs.

For PQDSC, we compare how good the algorithms approximate the Pareto-frontier for randomly generated registries and workflows. We implemented the genetic algorithm proposed by Claro et al. \[6\] (PGA in the following) using the JNSGA II Java library\[3\]. We introduced the Pareto error as criterion, how good the frontier is approximated. Unfortunately, we would need the real Pareto-frontier to calculate it. Instead, we run both algorithms on the same test case, both will return an approximated Pareto-frontier. Now we first assume the PGA found the real Pareto-frontier and calculate the Pareto error of RADO under this assumption. Then, we do the inverse. Figure 10(e) compares RADOs with PGA 100 (the strongest PGA) and Figure 10(f) RADOs with RADO 1 (the weakest RADO). The error of RADO is small in comparison while the processing time is several orders of magnitude smaller (see Figure 10(d)).

7. COMPARISON WITH RELATED WORK
We discuss these approaches to QDSC: Integer Linear Programming (ILP) \[11\] \[13\] \[1\], Genetic Algorithms (GA) \[8\] \[9\], and other heuristics (HEU) \[7\] \[2\] \[10\] \[4\] \[7\]. We use the three desirable theoretical properties we identified in the introduction as criteria for our comparison. Table 3 summarizes our results. An approach provides a feature if at least one of the cited representatives provides this feature. ILP guarantees

http://sourceforge.net/projects/jnsga2/
to find the optimal solution but solves NP-hard optimization problems and cannot guarantee polynomial run time (unless \( P = NP \)). We are not aware of any ILP based method for QDSC that approximates the Pareto-frontier. Heuristic approaches (GA, HEU) can guarantee polynomial run time with the right parameter setting but do not give formal approximation guarantees. Some of these approaches offer parameters to tune the approximation quality indirectly (e.g. number of generations for GA). However, the relation between a specific parameter setting and the approximation quality is not transparent and only of heuristic nature. Among the cited publications, only the approach by Claro et al. [6] approximates the Pareto-frontier.

8. CONCLUSION

We introduced the RADO (Recursive Assembly of Discretized Optima) algorithm for QDSC. It approximates the QoS Pareto-frontier with formal guarantees on approximation quality and in polynomial time. During our experimental evaluation, RADO outperformed classic approaches in terms of processing time and quality at the same time.

9. REFERENCES


