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) [11]. 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/

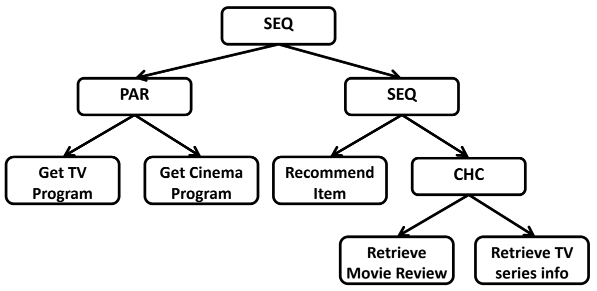


Figure 1: Example workflow

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. [3]. 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 $\mathbb{S}$ 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 binary tree $\langle V, E\rangle$ (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 $\mathit{isSimple}(v)$ 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 $\mathit{candidates}(v) \subseteq \mathbb{S}$. 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 $\mathit{construct}(v) \in \{SEQ, PAR, CHC\}$. The semantic of $\mathit{construct}(v) = SEQ$ 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 $b : \{v \in V : \mathit{isSimple}(v)\} \to \mathbb{S}$ that maps simple activities to concrete services such that for every simple activity v: $b(v) \in \mathit{candidates}(v)$. 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 5 simple and 4 complex activities. It first retrieves cinema and TV program in parallel by two service invocations. The retrieved

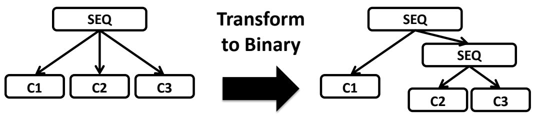


Figure 2: Transformation to binary tree

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 $\mathbb{A}$. We represent QoS values as positive real numbers. QoS values for all attributes can be represented as vectors within the *quality space* $\mathbb{Q} = \mathbb{R}_+^{|\mathbb{A}|}$. For $\vec{q} \in \mathbb{Q}$, we denote by $\vec{q}_a$ the component of the vector that refers to attribute $a \in \mathbb{A}$. 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 $\vec{AgQ}[v]$ 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 $\vec{AgQ}_a(v)$. 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 $\vec{Q}$ which estimates the QoS of every activity $v \in V$ for a binding b:

$$\vec{Q} : (v, b) \mapsto \begin{cases} \text{QoS for } b(v) \in \mathbb{S} & \text{if } \mathit{isSimple}(v), \\ \vec{AgQ}[v](\vec{Q}(v1), \vec{Q}(v2)) & \text{if } (v, v1), (v, v2) \in E. \end{cases} \tag{1}$$

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* $\vec{\sigma}$ maps from the QoS space $\mathbb{Q}$ to the *scaled quality space* $\mathbb{SQ} = [0, 1]^{|\mathbb{A}|}$. We scale with regards to a multi-dimensional *scaling range* which is defined by two QoS vectors, the *lower bound* $\vec{l} \in \mathbb{Q}$ and the *upper bound* $\vec{u}$. We define $\vec{\sigma}$ component-wise for every attribute a:

$$\vec{\sigma}_a : (\vec{q}_a, \vec{l}_a, \vec{u}_a) \mapsto \begin{cases} \dfrac{|[\vec{l}_a, \vec{u}_a] \cap [\vec{l}_a, \vec{q}_a]|}{\vec{u}_a - \vec{l}_a} & \text{if } a \text{ positive}, \\ \dfrac{|[\vec{l}_a, \vec{u}_a] \cap [\vec{q}_a, \vec{u}_a]|}{\vec{u}_a - \vec{l}_a} & \text{if } a \text{ negative}. \end{cases} \tag{2}$$

Example 2. Figure 3 shows how we scale response time to lower (l) and upper (u) bound. Note that a higher value in $\mathbb{Q}$ is scaled to a lower value in $\mathbb{SQ}$ since response time is

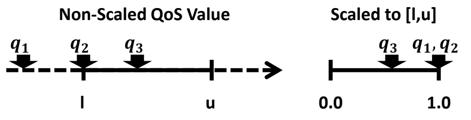


Figure 3: Scaling for response time QoS

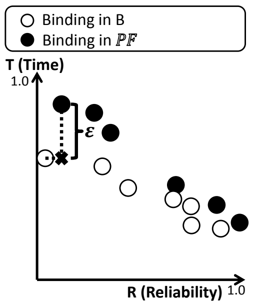


Figure 4: Calculating the Pareto error

a negative attribute. Also note that q_1 and q_2 are scaled to the same value since they only differ outside the interval $[l, u]$.

2.3 Pareto-Optimality

We compare bindings with regards to their scaled QoS values for some activity $v \in V$. Therefore, the following definitions of optimality are relative to the chosen lower and upper bound. We assume a fixed lower bound $\vec{l}$ and upper bound $\vec{u}$ and activity v in the following. We use short notations (only for this subsection): $\vec{Q}(v, b) = \vec{Q}(v)$ and $\vec{\sigma}(\vec{q}) = \vec{\sigma}(\vec{q}, \vec{l}, \vec{u})$. Let $b1$ and $b2$ two bindings for v. We say that $b1$ *dominates* $b2$ and write $b1 \sqsupseteq b2$ if $b1$ is at least as good in every QoS dimension ($\forall a \in \mathbb{A} : \vec{\sigma}(b1)_a \geq \vec{\sigma}(b2)_a$) and better in at least one dimension ($\exists a \in \mathbb{A} : \vec{\sigma}(b1)_a > \vec{\sigma}(b2)_a$). We denote by $\mathbb{B}_v$ the set of all possible bindings for activity v. The *Pareto-frontier* $\mathbb{PF}_v \subseteq \mathbb{B}_v$ for v is the (maximal) subset of all possible bindings that are not dominated by any other binding ($\forall b1 \in \mathbb{PF}_v \nexists b2 \in \mathbb{B}_v : b2 \sqsupseteq b1$). Our goal is to approximate this Pareto-frontier and we therefore need a measure on how closely a set of bindings approximates the Pareto-frontier. For a set of bindings $B \subseteq \mathbb{B}_v$ we define the *Pareto error* $PEr(B, v, \vec{l}, \vec{u})$ as the smallest $\varepsilon \in \mathbb{R}_+$ such that (3) holds.

$$\forall b1 \in \mathbb{PF}_v \exists b2 \in B \forall a \in \mathbb{A} : \vec{\sigma}(b1)_a - \vec{\sigma}(b2)_a \leq \varepsilon \qquad (3)$$

In other words, for every binding $b1$ on the Pareto-frontier, there is a binding $b2 \in B$ whose scaled QoS values are for no QoS dimension worse by more than ε comparing to $b1$.

Example 3. Figure 4 shows how the Pareto error is calculated. Bindings are represented as points within the scaled quality space (response time and reliability). The Pareto-frontier is closely approximated for bindings with good reliability. The approximation is not as good for bindings on the Pareto-frontier with low reliability (but good response time). This is where we have the biggest distance between approximation and Pareto-frontier, the Pareto error ε.

2.4 Problem statement

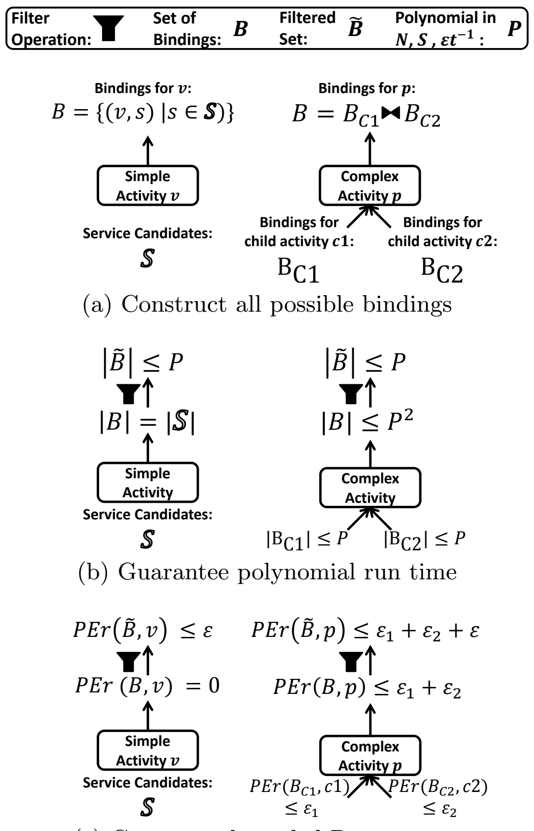
We define the problem of *Pareto Quality-Driven Service Composition (PQDSC)*. The goal of QDSC is to find bindings which maximize a utility function on their QoS values. The goal in PQDSC is to find sets of bindings (instead of single bindings) that approximate the Pareto-frontier. The input to PQDSC is the target precision εt and a tuple describing an abstract workflow, QoS aggregation for the activities and the available services:

$$I = \langle V, E, isSimple, candidates, construct, \vec{AgQ}, \mathbb{S} \rangle \qquad (4)$$

A solution to the PQDSC problem is a set of bindings B such that *i)* no binding $b1$ in B is dominated by another binding $b2$ in B ($\nexists b1, b2 \in B : b1 \sqsupseteq b2$), *ii)* the Pareto error of B for the root node $vr \in V$ is bounded by εt. The Pareto error is always defined with regards to a specific scaling range. We define the *total quality range* $\langle \vec{ql}, \vec{qu} \rangle$ for a workflow such that the QoS at vr of any possible binding b is included in this range: $\forall a \in \mathbb{A} : \vec{ql}_a \leq \vec{Q}(vr, b)_a \leq \vec{qu}_a$. 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 $\vec{ql}_a$ to the lower and $\vec{qu}_a$ to the upper bound of their a-priori value domain (e.g. for reliability $\vec{ql}_a = 0$, $\vec{qu}_a = 1$). Denote by $\mathbb{B}$ 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: $\vec{ql}_a = \min_{b \in \mathbb{B}} \vec{Q}(vr, b)_a$. We set the upper bound correspondingly: $\vec{qu}_a = \max_{b \in \mathbb{B}} \vec{Q}(vr, b)_a$. Note that those values can be calculated efficiently as shown by Zeng et al. [11]. We use $\vec{ql}$ and $\vec{qu}$ as defined here for the remainder of the paper. Now we can formulate requirement *ii)* on the result set B formally: $PEr(B, vr, \vec{ql}, \vec{qu}) \leq \varepsilon t$.

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, $S = \sum_{v \in V} |candidates(v)|$ (assuming $candidates(v) = \emptyset$ if $isSimple(v) = \textbf{false}$) 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 $A = |\mathbb{A}|$ we denote the number of attributes and by εt again the target precision. For our complexity analysis, we consider N, S and ε^{-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 [11]).



Filter Operation:
Set of Bindings: B
Filtered Set: $\tilde{B}$
Polynomial in $N, S, \varepsilon t^{-1}$: P
Bindings for v:
$B = \{(v,s) \mid s \in \mathbb{S}\}$
Bindings for p:
$B = B_{C1} \bowtie B_{C2}$
Simple Activity v
Complex Activity p
Service Candidates:
$\mathbb{S}$
Bindings for child activity $c1$:
Bindings for child activity $c2$:
B_{C1}
B_{C2}
$|\tilde{B}| \leq P$
$|B| = |\mathbb{S}|$
$|B| \leq P^2$
Simple Activity
Complex Activity
$|B_{C1}| \leq P$
$|B_{C2}| \leq P$
$PEr(\tilde{B}, v) \leq \varepsilon$
$PEr(B, v) = 0$
$PEr(\tilde{B}, p) \leq \varepsilon_1 + \varepsilon_2 + \varepsilon$
$PEr(B, p) \leq \varepsilon_1 + \varepsilon_2$
$PEr(B_{C1}, c1) \leq \varepsilon_1$
$PEr(B_{C2}, c2) \leq \varepsilon_2$

(a) Construct all possible bindings

(b) Guarantee polynomial run time

(c) Guarantee bounded Pareto error

Figure 5: Refinement of recursive procedure

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. For simple activities, every service candidate corresponds to one binding. Bindings for complex activities are constructed using the join operator.

Definition 1. **(Join Operator)** Let $c1, c2, p \in V$ and p is the parent activity of $c1$ and $c2$. Let $B1$ a set of bindings for $c1$ and $B2$ a set of bindings for $c2$. The *join* between $B1$ and $B2$ is a set of bindings for p which is defined by $B1 \bowtie B2 = \{b1 \cup b2 | b1 \in B1, b2 \in B2\}$.

We could use this algorithm to construct all possible bindings for the workflow and then filter bindings which are not Pareto-optimal. This would even guarantee a Pareto error of 0. Unfortunately, the number of possible bindings and the number of Pareto-optimal bindings can grow exponentially in N. We now show how to cope with this problem.

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 εt^{-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 $O(P^2)$, 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 εt^{-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* $\vec{cl}(v) \in \mathbb{Q}$ and the *critical upper bound* $\vec{cu}(v) \in \mathbb{Q}$ such that for the root activity $vr \in V$ $\vec{cl}(vr) = \vec{ql}$ and $\vec{cu}(vr) = \vec{qu}$ and the following requirement holds.

Requirement 3. Let $B1$ 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 $\varepsilon p = PEr(B1 \bowtie B2, p, \vec{cl}(p), \vec{cu}(p))$, the error for the joined set, $\varepsilon 1 = PEr(B1, c1, \vec{cl}(c1), \vec{cu}(c1))$, and $\varepsilon 2 = PEr(B2, c2, \vec{cl}(c2), \vec{cu}(c2))$. Then (5) holds.

$$\varepsilon p \leq \varepsilon 1 + \varepsilon 2 \tag{5}$$

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 ε 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: // $I = \langle V, E, isSimple, candidates, construct, \vec{AgQ}, \mathbb{S}\rangle$
 2: function RADO($I, \varepsilon t, v$)
 3:     $res \leftarrow \emptyset$ // Represents result set
 4:     // Test: simple or complex activity?
 5:     if $isSimple(v)$ then
 6:         // For all candidate services
 7:         for all $s \in candidates(v)$ do
 8:             $resItem \leftarrow (\{(v,s)\}, \vec{Q}(s))$
 9:             $res \leftarrow res \cup \{resItem\}$
10:         end for
11:     else
12:         // activity is complex
13:         $\{c1, c2\} \leftarrow \{c \in V | (v,c) \in E\}$
14:         $res1 \leftarrow$ RADO($I, \varepsilon t, c1$)
15:         $res2 \leftarrow$ RADO($I, \varepsilon t, c2$)
16:         $res \leftarrow$ join($res1, res2, \vec{AgQ}(v)$)
17:     end if
18:     return filter($res, v, \varepsilon t/|V|$)
19: end function
```

$v \in V$. Let $\widetilde{B}$ the result of filtering B with parameter ε. Then (6) must hold.

$$PEr(\widetilde{B}, v, \vec{cl}(v), \vec{cu}(v)) \leq PEr(B, v, \vec{cl}(v), \vec{cu}(v)) + \varepsilon \quad (6)$$

Figure 5(c) illustrates how the precision evolves when treating simple and complex activities. We perform N filtering operations and the accumulated Pareto error at the root activity will be $N \cdot \varepsilon$. We choose $\varepsilon = \varepsilon t \cdot N^{-1}$ to guarantee the upper bound on the Pareto error.

3.4 Pseudocode

We present the pseudocode for the procedure described before. Algorithm 1 is the main procedure. Input parameter I describes the abstract workflow with available services (according to (4)) and εt the upper bound on the Pareto error. $v \in V$ designates the activity (the node in the workflow tree) to be treated by this instance. For the initial call to *RADO*, we set v to the root node of $\langle V, E\rangle$. The output of the algorithm is a set of pairs $\{(b_1, \vec{q1}), (b_2, \vec{q2}), \ldots\}$ such that b_i is a binding for v and $\vec{qi}$ the QoS of this binding for v: $\vec{qi} = \vec{Q}(v, b_i)$. The final output (for the root node) respects the upper bound on the Pareto error and does not contain any two bindings such that one of them dominates the other. Algorithm 1 constructs bindings for simple activities from the service candidates and bindings for complex activities from the results of recursive calls. Note that we use in line 13 the fact that $\langle V, E\rangle$ is binary. The Join function (Algorithm 2) takes as input two sets of bindings with their QoS and joins them. It corresponds to the join operator (Def. 1), except that it additionally aggregates QoS (using the aggregation function represented by input parameter $\vec{F}$).We define the filtering function (call in line 18) in Section 4.

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.

Algorithm 2 Join two result sets

```
 1: function JOIN($res1, res2, \vec{F}$)
 2:     $res \leftarrow \emptyset$
 3:     for all $(b1, \vec{q1}) \in res1$ do
 4:         for all $(b2, \vec{q2}) \in res2$ do
 5:             $b \leftarrow b1 \cup b2$ // Union of bindings
 6:             $\vec{q} \leftarrow \vec{F}(\vec{q1}, \vec{q2})$ // Aggregate QoS
 7:             $res \leftarrow res \cup \{(b, \vec{q})\}$
 8:         end for
 9:     end for
10:     return $res$
11: end function
```

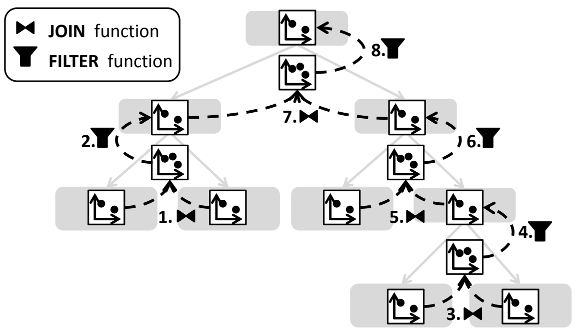


Figure 6: Function calls for example workflow

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 $\mathbb{SQ}$ to a finite, discrete space $\mathbb{DQ}$, the *discretized quality space.* Discretization depends on a parameter ε. A higher ε means lower precision during discretization and $\mathbb{DQ} = \{0, \ldots, \lfloor 1/\varepsilon \rfloor\}^A$. We define the *scaling function* $\vec{\delta}[\varepsilon] : \mathbb{SQ} \rightarrow \mathbb{DQ}$ for $\vec{sq} \in \mathbb{SQ}$:

$$\vec{\delta}[\varepsilon](\vec{sq})_a \mapsto \lfloor \vec{sq}_a / \varepsilon \rfloor \quad (7)$$

We can translate our definition of dominance between bindings to the discretized space. Denote by $\vec{dq1}, \vec{dq2} \in \mathbb{DQ}$ the discretized (with precision ε), scaled QoS of bindings $b1$ and $b2$ for activity $v \in V$ (both scaled to $\vec{cl}(v), \vec{cu}(v)$). We write $b1 \sqsupset_\varepsilon b2$ if $\forall a \in \mathbb{A} : \vec{dq1}_a \geq \vec{dq2}_a$ and $\exists a \in \mathbb{A} : \vec{dq1}_a > \vec{dq2}_a$. While filtering a set of bindings B, we remove a binding $b1 \in B$ if there exists a binding $b2 \in B$ such that $b2 \sqsupset_\varepsilon b1$. 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 ε than $b1$ in every QoS dimension. Therefore, the Pareto error may at most increase by ε and Requirement 4 is satisfied. Our filtering could naively be implemented by comparing all bindings pair-wise. This satisfies the requirements on the time complexity as speci-

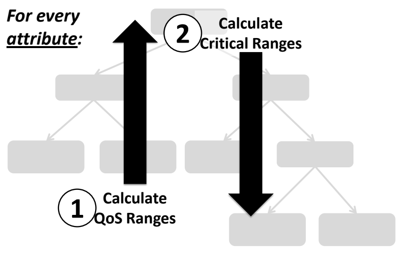


Figure 7: Calculating critical ranges

Table 1: Classification of QoS attributes

Type	Value Domain	Aggregation Functions	Examples
1	$[0,1]$	$\min, \max, \sum, \prod$	Reliability, availability
2	[0,qu]	$\min, \max, \sum$	Reputation
3	$[0,\infty]$	$\min, \sum$	Throughput
4	$[0,\infty]$	$\max, \sum$	Time, cost

fied in Requirement 1 (we show a more efficient method in Section 4.2). We keep at most one binding for every vector in the discretized space. The number of different vectors in $\mathbb{DQ}$ is restricted by $((\varepsilon+1)^{-A})$ and therefore polynomial in ε^{-1} (as A is constant). This satisfies Requirement 2.

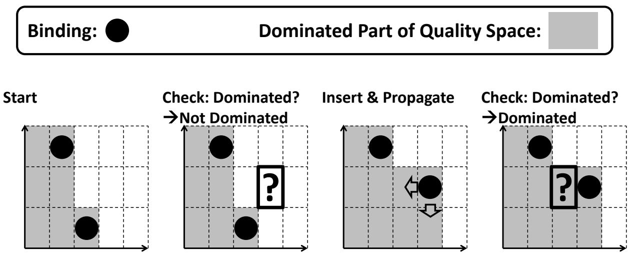
4.1 Constructing the critical range

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 from 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 child activity of complex activity $p \in V$. Fix an attribute $a \in \mathbb{A}$. For the remainder of this Subsection we omit attribute index and vector arrows and implicitly refer to this attribute (e.g. $cl(c) = \vec{cl}_a(c)$). 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 5 that this definition of critical ranges guarantees that Requirement 3 is satisfied.

Table 2: Definition of critical ranges

Attr. Type	Agg. Fct.	Type D
1,2	(All)	cl(c)=ql(c) cu(c)=qu(c)
3	min	$cl(c) = ql(c)$ $cu(c) = cu(p)$
	$\sum$	$cl(c) = ql(c)$ $cu(c) = ql(c) + \frac{cu(p) - cl(p)}{W}$
4	max	$cl(c) = cl(p)$ $cu(c) = qu(c)$
	$\sum$	$cl(c) = qu(c) - \frac{cu(p) - cl(p)}{W}$ $cu(c) = qu(c)$



Binding: ●
Dominated Part of Quality Space: ▨
Start
Check: Dominated? →Not Dominated
Insert & Propagate
Check: Dominated? →Dominated
?
?

Figure 8: Illustration of filtering

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 $\vec{cl}(v)$ and $\vec{cu}(v)$ (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 ε. Note that we call the filtering function with $\varepsilon = \varepsilon t/N$ 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 $\mathbb{DQ}$. 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 $\vec{\delta}$ 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

Algorithm 3 Filter out dominated and equivalent bindings

```
1: function FILTER($toFilter, v, \varepsilon$)
2:     // Scaled, discretized QoS - index space for arrays
3:     $\mathbb{DQ} \leftarrow \{0, \ldots, \lfloor 1/\varepsilon t \rfloor\}^{A}$
4:     // Initialise arrays
5:     for all $\vec{i} \in \mathbb{DQ}$ do
6:         $dominated[\vec{i}] \leftarrow$ false
7:         $itemsTable[\vec{i}] \leftarrow \emptyset$
8:     end for
9:     // Filter bindings
10:    for all $(b, \vec{q}) \in toFilter$ do
11:        $\vec{sq} \leftarrow \vec{\sigma}(\vec{q}, \vec{cl}(v), \vec{cu}(v))$ // scale to critical range
12:        $\vec{i} \leftarrow \vec{\delta}[\varepsilon](\vec{sq})$ // discretize
13:        if $dominated[\vec{i}] =$ false then
14:            propagate($dominated, itemsTable, \vec{i}$)
15:            $itemsTable[\vec{i}] \leftarrow (b, \vec{q})$
16:        end if
17:    end for
18:    $res \leftarrow \emptyset$
19:    // Retrieve non-dominated items from table
20:    for all $\vec{i} \in \mathbb{DQ}$ do
21:        $res \leftarrow res \cup \{itemsTable[\vec{i}]\}$
22:    end for
23:    return $res$
24: end function
```

Algorithm 4 Mark fields in QoS space as dominated

```
1: function PROPAGATE($dominated, itemsTable, \vec{i}$)
2:     $itemsTable[\vec{i}] \leftarrow \emptyset$
3:     if $dominated[\vec{i}] =$ false then
4:         $dominated[\vec{i}] \leftarrow$ true
5:         for all $a \in \mathbb{A}$ do
6:             $\vec{j} \leftarrow \vec{i}$
7:             $\vec{j}_a \leftarrow \vec{j}_a - 1$
8:             if $\vec{j}_a \geq 0$ then
9:                 propagate($dominated, itemsTable, \vec{j}$)
10:            end if
11:        end for
12:    end if
13: end function
```

treatment of two consecutive bindings. The first binding is not dominated by the previously inserted bindings and is therefore inserted. The corresponding fields are marked as dominated and the dominated bindings are deleted. The next binding would be dominated and is not inserted.

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

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 1 is executed (input parameter v is the root activity for the first call). Algorithm 1 uses Algorithm 2 and Algorithm 3 as auxiliary functions. Algorithm 3 uses Algorithm 4 as sub-procedure.

5.2 Approximation precision

We fix tree arbitrary activities $c_1, c_2, p \in V$ 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 $F = \vec{AgQ}[p]_a$, by σ_p the scaling function for the parent $\sigma_p(q) = \vec{\sigma}(\vec{q}, \vec{cl}(p), \vec{cu}(p))_a$, by σ_1 the scaling function for c_1, $\sigma_1(q) = \vec{\sigma}(\vec{q}, \vec{cl}(c), \vec{cu}(c))_a$, 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: $PEr(B, v) = PEr(B, v, \vec{cl}(v), \vec{cu}(v))$. The following theorem provides a reformulation of Requirement 3 that we use for constructing critical ranges.

THEOREM 1. *If (8) is satisfied for all attributes $\forall a \in \mathbb{A}$, all possible QoS values $q_1, q_2 \in \mathbb{R}_+$ and for all $\varepsilon \in \mathbb{R}$ with $0 \leq \varepsilon \leq qu(c_1) - q_1$, then the propagation of the Pareto error is bounded and (9) holds.*

$$
\begin{aligned}
&|\sigma_p(F(q_1 + \varepsilon, q_2)) - \sigma_p(F(q_1, q_2))| \\
&\quad \leq |\sigma_1(q_1 + \varepsilon) - \sigma_1(q_1)| \qquad (8) \\
&PEr(B1 \bowtie B2, p) \leq PEr(B1, c_1) + PEr(B2, c_2) \qquad (9)
\end{aligned}
$$

PROOF. Set $\varepsilon p = PEr(B1 \bowtie B2, p)$, $\varepsilon 1 = PEr(B1, c_1)$ and $\varepsilon 2 = PEr(B2, c_2)$. Let $\tilde{b}$ a Pareto-optimal binding for p. Assume that $\tilde{b}$ is not Pareto-optimal for the child activities c_1 and c_2 (or both). Then we can always find a binding $\widetilde{bo}$ with equivalent QoS in p ($\vec{Q}(p, \tilde{b}) = \vec{Q}(p, \widetilde{bo})$) and $\widetilde{bo}$ 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 $\widetilde{bo} = \widetilde{b1} \cup \widetilde{b2}$ such that $\widetilde{b1}$ is a Pareto-optimal binding for c_1 and $\widetilde{b2}$ Pareto-optimal for c_2. Let $b1 \in B1$ a binding for c_1 whose scaled QoS value is not worse by more than $\varepsilon 1$ in every QoS dimension comparing with $\widetilde{b1}$ (such a binding must exist since $PEr(B1, c_1) = \varepsilon 1$). Let $b2 \in B2$ a binding for c_2 whose scaled QoS value is not worse by more than $\varepsilon 2$ in every QoS dimension comparing with $\widetilde{b2}$. Now imagine that we change the binding for p in two steps from $b1 \cup b2$ to $\widetilde{b1} \cup \widetilde{b2}$: first, we replace $b1$ by $\widetilde{b1}$, then we replace $b2$ by $\widetilde{b2}$. Fix an arbitrary attribute $a \in \mathbb{A}$ and denote $q_1 = \vec{Q}(c_1, b1)_a$, $\widetilde{q_1} = \vec{Q}(c_1, \widetilde{b1})_a$, $q_2 = \vec{Q}(c_2, b2)_a$, and $\widetilde{q_2} = \vec{Q}(c_2, \widetilde{b2})_a$. Further, let

$$\Delta 1 = \sigma_p(F(\widetilde{q_1}, q_2)) - \sigma_p(F(q_1, q_2)) \quad (10)$$

$$\Delta 2 = \sigma_p(F(\widetilde{q_1}, \widetilde{q_2})) - \sigma_p(F(\widetilde{q_1}, q_2)) \quad (11)$$

We have

$$\sigma_p(F(\widetilde{q_1}, \widetilde{q_2})) - \sigma_p(F(q_1, q_2)) = \Delta 1 + \Delta 2 \quad (12)$$

Now if $\widetilde{q_1} \leq q_1$ then $\Delta 1 \leq 0 \leq \varepsilon 1$. If $\widetilde{q_1} > q_1$ then $\Delta 1 = |\Delta 1|$ and applying (8) yields $|\Delta 1| \leq |\sigma_1(\widetilde{q_1}) - \sigma_1(q_1)|$ and again $\Delta 1 \leq \varepsilon 1$. By a similar reasoning, we find that $\Delta 2 \leq \varepsilon 2$. Therefore, (9) holds. $\square$

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 2($qu(v) = cu(v) = cu$ and $ql(v) = cl(v) = cl$ 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 σ for our specific choice of critical ranges. First, note that QoS values are always contained within the critical range. Therefore, $|[cl, q] \cap [cl, cu]| = q - cl$. Also, $cu(v) - cl(v) = cu - cl$ is constant over all activities $v \in V$. Hence

$$\sigma_1(q) = \sigma_p(q) = \sigma(q) = \frac{q - cl}{cu - cl} \quad (13)$$

So σ is a linear transformation and (14) and (15) hold.

$$\begin{aligned}\sigma(F(q_1 + \varepsilon, q_2)) - \sigma(F(q_1, q_2)) \\ = \sigma(F(q_1 + \varepsilon, q_2) - F(q_1, q_2)) \quad (14)\end{aligned}$$

$$\sigma(q_1 + \varepsilon) - \sigma(q_1) = \sigma(\varepsilon) \quad (15)$$

σ is monotone (for positive attributes), so we just have to show (16) which implies (8).

$$F(q_1 + \varepsilon, q_2) - F(q_1, q_2) \leq \varepsilon \quad (16)$$

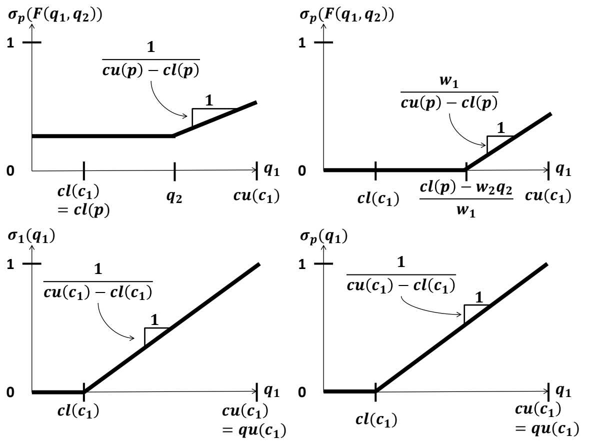
Equation (16) is obviously satisfied for minimum, maximum, and the weighted sum (with weights between 0 and 1). It is satisfied for the product, because in this case the attribute is of type 1, so $cl = 0$, $cu = 1$ and $q_2 \leq 1$ which implies (17).

$$F(q_1 + \varepsilon, q_2) - F(q_1, q_2) = \varepsilon \cdot q_2 \leq \varepsilon \quad (17)$$

$\square$

Theorem 3. *Assume an attribute with* ***unbounded domain*** *(type 3 or 4) and the critical ranges are defined according to Table 2. Then this implies (8).*

Proof. We assume that the attribute is positive and of type 4. Equation (8) compares $\sigma_1(q_1)$ 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 9(a). The curve for σ_1 (lower half of Figure 9(a)) follows directly from the definition of σ_1 and because we assume



(a) F is the maximum (b) F is the weighted sum

Figure 9: Propagation of QoS changes

that a is positive. We explain the curve for $\sigma_p(F)$. Imagine we grow q_1 from 0 to $cu(c_1)$. As long as $q_1 \leq cl(c_1)$, F is either constant ($F = q_2$) or $F(q_1, q_2) = q_1 \leq cl(p)$. In both cases, $\sigma_p(F)$ remains constant. If $cl(c_1) \leq q_1 \leq q_2$ (this case may not exist if $q_2 < cl(c_1)$) then F (and therefore $\sigma_p(F)$) remain constant, too. If $q_1 > q_2$ then F grows with slope 1 and $\sigma_p(F)$ grows with slope $1/(cu(p) - cl(p))$. σ_1 grows with slope $1/(cu(c_1) - cl(c_1))$ for this domain for q_1. According to the definition of critical ranges: $cl(p) = cl(c_1)$. We also have $qu(p) = \max(qu(c_1), qu(c_2)) \geq qu(c_1)$. Therefore the slope of σ_1 is greater or equal to the slope of $\sigma_p(F)$ for all values of q_1 and q_2. Therefore, the difference $\sigma_1(q_1 + \varepsilon) - \sigma_1(q_1)$ is always greater or equal to $\sigma_p(F(q_1 + \varepsilon, q_2)) - \sigma_p(F(q_1, q_2))$ and (8) is always satisfied.

Now assume $F = w_1 \cdot q_1 + w_2 \cdot q_2$ (the weighted sum). Figure 9(b) shows the development of σ_1 and $\sigma_p(F)$. We grow q_1 from 0 to $cu(c_1)$ again. As long as $F(q_1, q_2) \leq cl(p)$, it is $\sigma_p(F) = 0$. We have $F(q_1, q_2) \geq cl(p)$ if and only if $q_1 \geq (cl(p) - w_2 \cdot q_2)/w_1$. We now show that the point (the value for q_1) where σ_1 starts to grow is the same or left from the point where $\sigma_p(F)$ starts to grow. By solving the equation for $cl(c)$ from Table 2 for $cl(p)$, we obtain

$$cl(p) = cu(p) + w_1 \cdot (cl(c_1) - qu(c_1)) \quad (18)$$

This implies

$$\frac{cl(p) - w_2 \cdot q_2}{w_1} = \frac{cu(p)}{w_1} + (cl(c_1) - qu(c_1)) - \frac{w_2}{w_1} q_2 \quad (19)$$

Using $cu(p) = qu(p) = w_1 qu(c_1) + w_2 qu(c_2)$ and $q_2 \leq qu(c_2)$ in (19) shows

$$\frac{cl(p) - w_2 \cdot q_2}{w_1} \geq cl(c_1) \quad (20)$$

We compare the slopes from σ_1 and $\sigma_p(F)$ once they are not zero anymore. For σ_1 the slope is $1/(cu(c_1) - cl(c_1))$. For $\sigma_p(F)$ the slope is $w_1/(cu(p) - cl(p))$. We have

$$cu(p) - cl(p) = w_1(cu(c_1) - cl(c_1)) \quad (21)$$

and therefore the slope of σ_1 is always greater or equal to the slope of $\sigma_p(F)$. Applying the same reasoning as before, this proves (8). So far we assumed a positive attribute of 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 εt^{-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 εt^{-1}). 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 $(N/\varepsilon t)^{A-1}$ ($A-1$ 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 2 is therefore in $O((N/\varepsilon t)^{2A-2})$. For complex activities, Algorithm 3 is called on the result of the join operation. Checking one binding (whether it should be inserted) and inserting it in *itemsTable* is in $O(1)$. However, in the worst case all cases in array *dominated* have to be marked by Algorithm 4. There are $(N/\varepsilon t)^{A}$ cases. In total, the filtering for one complex activity is performed in $O((N/\varepsilon t)^{2A-2}+(N/\varepsilon t)^{A})$. The filtering for a simple activity is performed in $O(S+(N/\varepsilon t)^{A})$. The total time complexity of *RADO* is

$$O(N\cdot(S+(N/\varepsilon t)^{A}+(N/\varepsilon t)^{2A-2})) \qquad (22)$$

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 1 is executed once for every activity. It first constructs a set of bindings. For simple activities, the constructed set of bindings is in $O(S)$ space. For complex activities, two sets of filtered bindings have to be stored ($res1$ and $res2$) which are in $O((N/\varepsilon t)^{A-1})$ space. Additionally, a joined set of bindings is constructed which is in $O((N/\varepsilon t)^{2A-2})$ 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 3.4 in order to improve the readability. We integrate Algorithm 2 and Algorithm 3 directly into 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 $(N/\varepsilon t)^{A-1}$ 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 *dominated* and *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

$$O(N\cdot(S+(N/\varepsilon t)^{A})) \quad \text{(space)} \qquad (23)$$

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 ε (instead of $\varepsilon t/N$). 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.

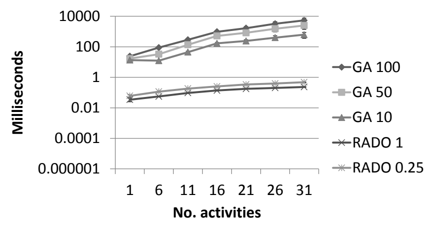
In **QDSC**, the goal is to find a binding that maximizes the utility function and respects the constraints. The utility function is a weighted sum over the scaled QoS, constraints correspond to lower bounds on the scaled QoS values. We compare RADO with the genetic algorithm (GA) by Canfora et al. [5]. 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 registry). 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[4]. 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) PGAs 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)).

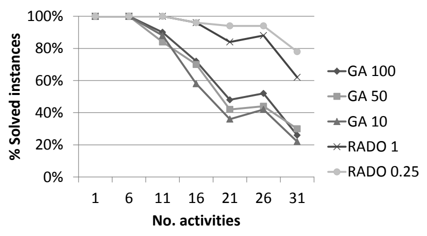
7. COMPARISON WITH RELATED WORK

We discuss these approaches to QDSC: Integer Linear Programming (ILP) [11, 3, 1], Genetic Algorithms (GA) [5, 6], 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

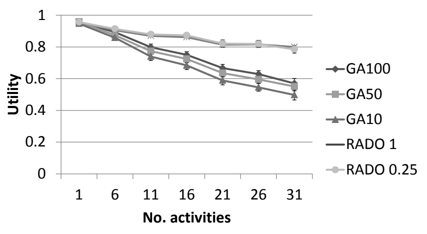
[4]http://sourceforge.net/projects/jnsga2/



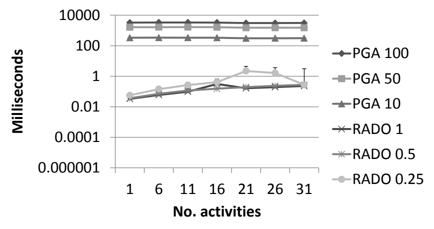
(a) QDSC - Processing time



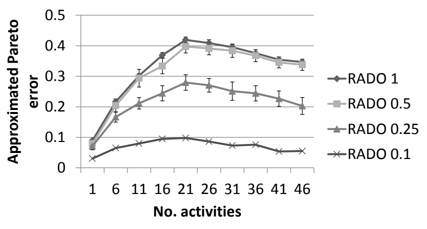
(b) QDSC - Solved instances



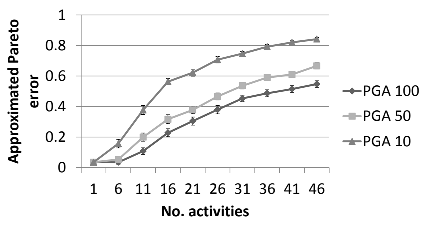
(c) QDSC - Utility



(d) PQDSC - Processing time



(e) PQDSC - Pareto error of RADO



(f) PQDSC - Pareto error of PGAs

Figure 10: Experimental evaluation

Table 3: Comparing features of QDSC methods

Method	Polynomial Run Time	Approximation Guarantees	Approx. Pareto Set
ILP	✗	✔	✗
GA	✔	✗	✔
HEU	✔	✗	✗
RADO	✔	✔	✔

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

[1] R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint driven web service composition in meteor-s. 2004.

[2] D. Ardagna and B. Pernici. Global and local qos guarantee in web service selection. In *Business Process Management Workshops*, pages 32–46. Springer, 2006.

[3] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, pages 369–384, 2007.

[4] R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz. Heuristics for QoS-aware web service composition. In *Int. Conf. on Web Services*, pages 72–82. IEEE, 2006.

[5] G. Canfora, M. Di Penta, R. Esposito, and M. Villani. An approach for QoS-aware service composition based on genetic algorithms. In *Conf. on Genetic and evolutionary computation*, pages 1069–1075. ACM, 2005.

[6] D. Claro, P. Albers, and J. Hao. Selecting web services for optimal composition. In *ICWS International Workshop on Semantic and Dynamic Web Processes, Orlando-USA*. Citeseer, 2005.

[7] D. Comes, H. Baraki, R. Reichle, M. Zapf, and K. Geihs. Heuristic Approaches for QoS-Based Service Selection. In *Int. Conf. on Service-Oriented Computing*, pages 441–455. Springer, 2010.

[8] E. Masri and Q. Mahmoud. Discovering the best web service. 2007.

[9] H. Yu, N. Benn, S. Dietze, C. Pedrinaci, D. Liu, J. Domingue, and R. Siebes. Two-staged approach for semantically annotating and brokering tv-related services. In *2010 IEEE International Conference on Web Services*, pages 497–503. IEEE, 2010.

[10] T. Yu and K. Lin. Service selection algorithms for composing complex services with multiple QoS constraints. In *Int. Conf. on Service-Oriented Computing*, pages 130–143. Springer, 2005.

[11] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.