An End-to-End Approach for QoS-Aware Service Composition

Florian Rosenberg, Predrag Celikovic, Anton Michlmayr, Philipp Leitner and Schahram Dustdar
Distributed Systems Group, Technical University Vienna
Argentierstrasse 8/184-1, Vienna, Austria
lastname@infosys.tuwien.ac.at

***Abstract*—A simple and effective composition of software services into higher-level composite services is still a very challenging task. Especially in enterprise environments, Quality of Service (QoS) concerns play a major role when building software systems following the Service-Oriented Architecture (SOA) paradigm. In this paper we present a composition approach based on a domain-specific language (DSL) for specifying functional requirements of services and the expected QoS in form of constraint hierarchies by leveraging hard and soft constraints. A composition runtime will resolve the user's constraints to find an optimized composition semi-automatically. To this end we leverage data flow analysis to generate a structured composition model and use two different techniques for the optimization, a constraint programming and an integer programming approach.**

I. INTRODUCTION

In the last few years, Service-Oriented Computing (SOC) is gaining momentum as a means to develop applications based on the Service-Oriented Architecture (SOA) paradigm [1]. The core entities of each SOA are loosely-coupled software services implementing parts of the main business functionality. These business services, combined with services from other business partners or public services, can be orchestrated into composite services, a task generally known as service composition. Currently, the process of composing services is mainly static by wiring together several services that are enacted using a composition engine. However, in distributed environments a static composition may lead to inflexible and non-adaptable application architectures. Existing services may raise errors or become unavailable and need to be replaced by other services resulting in a better performance or lower cost. Quality of Service (QoS) is an emerging category within SOC to express non-functional quality attributes of a service, such as the response time, cost or the supported security protocols. QoS is increasingly important when composing services because a degrading QoS in one of the services can negatively affect the QoS of the overall composition.

Most existing approaches lack a coherent framework for the specification, optimization as well as the generation of an executable composition [2–5]. Contrary to existing approaches which only consider hard constraints during the optimization, the approach presented in this paper allows to use both, hard and soft constraints to specify QoS. The former indicate required constraints that have to be satisfied during the composition process whereas the latter represent optional QoS constraints that are "nice to have" and should be satisfied if possible. To this end, the proposed composition approach does not optimize to find the best solution overall, however, it tries to find an optimal solution within the constraint boundaries given by the user as part of the composite service specification. In this paper our contribution is twofold: Firstly, we focus on the specification aspect of QoS-aware compositions on a microflow level, i.e., fine-grained short-running composite services, more on a technical level that implement parts of a business process or a so-called macroflow [6]. In particular, we describe VCL (Vienna Composition Language), a simple domain-specific language for specifying QoS-aware service compositions. In this language, developers can specify what functional and non-functional constraints (i.e., QoS) each service in a composition has to fulfill. QoS can be specified for the overall composition as well as for single services by using constraint hierarchies to express a fine-grained distinction of the importance of a constraint. Secondly, we combine a set of techniques and algorithms to transform and optimize a composition specified in VCL: (a) we leverage and extend an existing approach [7] to generate a structured composition model from the unstructured specification in VCL by analyzing the dataflow; (b) we optimize the structured composition model from step (a) in terms of QoS by transforming it into both, a constraint optimization problem (COP) and a linear integer programming (IP) problem. The optimization process will assign those service candidates to each activity in the composition that fulfills all the required constraints and the highest number of optional constraints in the hierarchy. If required constraints cannot be fulfilled and no valid composition can be generated, the developer is asked to relax some of the constraints and re-execute the aforementioned steps. The approach is evaluated in terms of its performance to demonstrate that the optimization is feasible to form the basis for design-time composition and runtime re-composition.

This paper is organized as follows: Section II provides a short overview of the Composition as a Service (CAAS) approach and its foundations. Section III describes the domain-specific composition language VCL. In Section IV we describe our composition and optimization approach followed by an evaluation in Section V. Section VI discusses the presented approach by highlighting some of its benefits and applicability, whereas Section VII describes some related work. Finally, Section VIII concludes this work and highlights some future work.

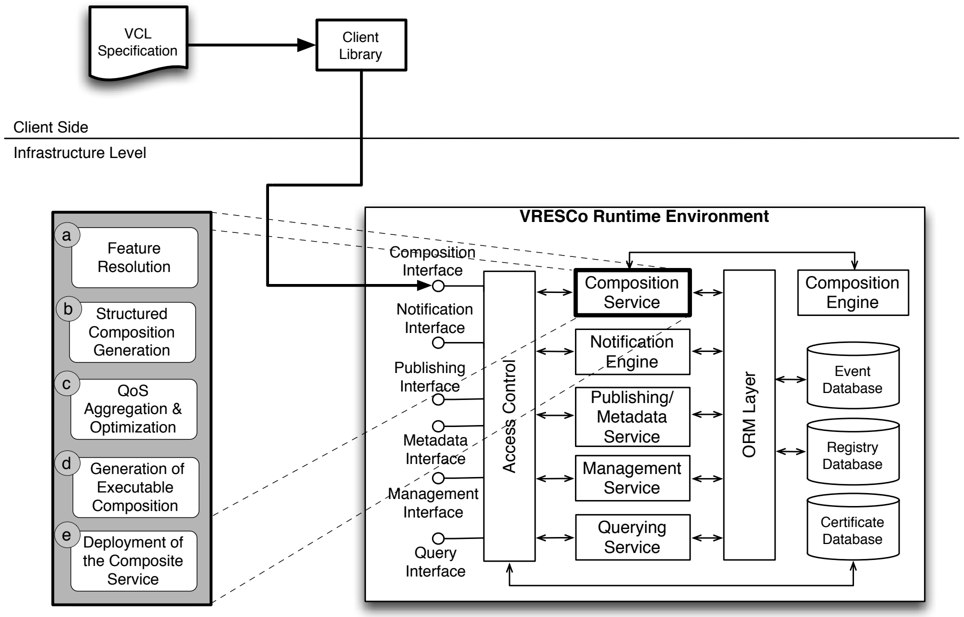


Fig. 1. Architectural Overview of Composition as a Service with VCL

II. Overview

One of the most important aspects when performing service composition in general, and QoS-aware composition in particular, is the possibility to coherently specify, optimize and deploy a composite service without the need to configure and setup all the infrastructure required to deploy and run a composite service. Therefore, our architectural approach follows service-oriented principles by providing "Composition as a Service" (CaaS) [8]. The idea is to provide composition as a hosted environment to reduce the need to install and maintain a composition infrastructure, thus it allows to specify, generate and deploy compositions on-the-fly. This principle is well-aligned with recent IT trends in utility computing such as Infrastructure-as-a-Service and Cloud Computing in general [9]. The basic architecture is illustrated in Figure 1 specifically showing the contribution of this paper, the *Composition Service* and its algorithms, on the left-hand side (Infrastructure Level) and the existing VRESCo runtime environment on the right-hand side (which is briefly explained below).

A. VRESCo Runtime

The VRESCo project (Vienna Runtime Environment for Service-Oriented Computing) [10] is a novel runtime and programming model based on an extensible service metadata model [11]. It addresses typical software engineering related issues in SOC, such as publishing services, dynamic binding and invocation as well as service discovery by using a type-safe query mechanism. Services and associated metadata are published into a registry database which is accessed using an ORM (Object Relational Mapping) layer. The query language VQL (similar to the Hibernate Query Language) is used to query all information stored in this database [10], whereas the event notification engine is responsible for publishing events when certain situations occur (e.g., a new service is published, QoS changes, etc.) [12]. The VRESCo core services are accessed either directly using SOAP or via the client library which provides a simple API. Furthermore, it offers mechanisms to dynamically bind and invoke services using the integrated Daios framework [13]. Finally, a QoS monitor [14] has been integrated to continuously measure the QoS attributes (e.g., response time, throughput, etc.) of the services. A detailed description of VRESCo is out of scope of this paper, however, the respective parts of the VRESCo project are described where needed to understand the approach presented in this paper. We refer the reader to the other papers on selected aspects of the VRESCo project.

B. Composition as a Service

By leveraging VRESCo's core services (such as publishing, querying, metadata), we integrate a *Composition Service* on the infrastructure level to encapsulate the overall composition approach. The user interacts with the system only by specifying a composition using a constraint-based composition language called VCL. This language allows to specify constraints on inputs, outputs, pre- and postconditions, and QoS. As mentioned above, we apply the theory of constraint hierarchies [15] for specifying the desired QoS for a service in the composition (local constraint) or the overall composition (global constraint). Formally, a constraint hierarchy H is a multiset of labeled constraints. H_0 denotes the required constraints in H. The sets H_1, H_2,…,H_n are defined for the hierarchy levels 1,2,…,n representing the optional constraints with different strengths. Each level expresses constraints that are equally important. In our approach the hierarchy levels are labeled $\{required, strong, medium, weak\}$ but this can be adapted to represent different hierarchy levels. When specifying QoS requirements for composite services, constraint hierarchies are very useful because many QoS requirements are “nice to have” but not necessarily required. Therefore, the composition process is more flexible compared to a model where only hard constraints are supported.

In our approach, the *Composition Service* can be accessed by using the VRESCo client library. Upon reception of a composition request, the composition service generates a composite service from the initial VCL description by performing tasks (a)–(e) from Figure 1. When a composite service can be generated, it will be deployed to the underlying composition engine and the newly created endpoint of the composite service will be returned to the caller. Besides describing the composition language VCL, we focus on steps (b)–(c); step (a) is briefly explained for the sake of completeness. Steps (d) and (e) involve the automated generation and deployment of the executable composition based on the outcome of the QoS optimization. Due to space restrictions, a detailed description of these parts is out of scope of this paper and described in more detail in [16].

III. Vienna Composition Language

The main idea behind VCL is to provide a simple domain-specific language for specifying QoS-aware composite services. The language follows a constraint-based approach in the sense that all functional and non-functional aspects of a composition are declared as constraints on *features* that should be composed. This notation is based on VRESCo’s metadata model [11], which abstracts from concrete services and its operations by using the notation of *features* that can be grouped into *categories*. For example, one can define a category `Hotel Services` with a feature `BookHotel` as part of the application metadata. At publishing time of a certain hotel booking service, the mapping from a concrete service operation to the feature has to be specified including the mapping of input and output to data concepts in the metadata model. The VRESCo publishing and metadata services provide the necessary functionality. The main reason for implementing such a model is to achieve fully transparent dynamic binding and invocation of a service (including mediation if the data of a service do not match with the feature in- and output). For example, a user can issue a query for services implementing the `BookHotel` feature and dynamically bind to one of the services found by the query without bothering about the concrete service details.

This feature-driven metadata model is leveraged by VCL to describe the core entities that are composed. Basically, each VCL specification consists of three major blocks: *feature definition*, (global and feature) *constraints* and the *business protocol*. The following EBNF grammar rules show the formal specification of the core parts. We do not depict all terminal symbols in details due to space restrictions[1].

[1]A work-in-progress version of the full grammar in EBNF and MGrammar notation is available at http://www.infosys.tuwien.ac.at/staff/rosenberg/vresco/.

```
<Composition> ::= composition <Name>;
                  <Feature> { <Feature> }
                  <GlobalConstraint>
                  { <FeatureConstraint> }
                  <BusinessProtocol>
<Feature> ::= feature <Alias>,
              <CategoryName>.<FeatureName>;
<GlobalConstraint> ::= constraint global
                       "{" <ConstraintBody> "}";
<FeatureConstraint> ::= constraint <Alias>
                        "{" <ConstraintBody> "}";
<ConstraintBody> ::= <InputConstraint>
    <OutputConstraint> <PrecondConstraint>
    <PostcondConstraint> <QoSConstraint>
```

Each constraint type (global and local) imposes restrictions on the input and output, pre- and postconditions and on QoS (cf., `<ConstraintBody>`). Input and output statements of a global constraint can be used to define the interface of the composed service, whereas input and output statements on a feature are used to constrain the feature input and output that is required in the composition. The final part is the specification of the business protocol (`<BusinessProtocol>`) describing how the services should be invoked and which data is assigned to the services. This is achieved by allowing to specify typical control flow constructs such as conditionals (`check` statement), loops (`while` statement) and a statement to invoke a service (`invoke` statement). For a complete list of statements please refer to the VCL specification as mentioned above.

```
1   composition tripplanner;
2
3   feature sf, *.SearchFlight;
4   feature sh, *.SearchHotel;
5   feature bf, *.BookFlight;
6   feature bh, *.BookHotel;
7
8   constraint global {
9     qos = {
10      responseTime = 5000, required;
11      availability = 0.95, strong;
12      price = 0.5, weak;
13    }
14  }
15
16  constraint sf {
17    input = {
18      SearchFlightRequest[
19        # define required members
20      ]
21    }
22    output = {
23      # define required output
24    }
25  }
26
27  invoke sf {...};
28  invoke sh {...};
29
30  check (sf.Flights.Count > 0 & sh.Hotels.Count > 0) {
31    invoke bf {...};
32    invoke bh {...};
33  }
34
35  return {
36    BookingResponse [
37      #assign result
38    ]
39  }
```

Listing 1. VCL Sample

Listing 1 shows features and global constraints of a simplified trip planner composite service (lines 3–14). In our approach, we assume that each feature is defined in the

VRESCo metadata model and concrete services are mapped to that feature. Several QoS attributes can be used to constrain features with non-functional requirements, either globally or for a single feature. Please refer to Table I for a subset of QoS attributes currently supported in VCL. By specifying `responseTime = 5000` (line 10) we constrain the response time of the overall composition to ≤ 5000. We simply use the = character instead of $\leq$ because the response time is a QoS attribute with descending dimension in the sense that a lower value is generally better (the same holds for `price`). For the `availability` attribute it is vice-versa (it has ascending dimension meaning that a higher value is better). Additionally, we set the response time constraint strength to `required` to express that this constraint has to hold. The two other constraints are optional with different strength values. A feature constraint for expressing constraints for input and output of a given feature is given from lines 16–25. In this case we look for a flight search service with a specific feature input and output. From lines 27–39, the business protocol specification for the trip planner is specified. Both search features (`sf` and `sh`) are invoked followed by a simple check whether some flights and hotels are found. If true, both booking features (`bf` and `bh`) are invoked. At the end, a `BookingResponse` message is returned containing information about the booking. In this example we omit data assignment in the empty curly braces for readability. The business protocol specification does not have an explicit notation for specifying which parts are executed sequentially and which parts can be executed in parallel (i.e., we do not have AND-splits, XOR-splits as explicit constructs [17] as many graph-based approaches). Such an unstructured composition approach enables a simpler specification from a user perspective, however, it may lead to errors during the execution (e.g., deadlocks) if not properly validated. Thus, a transformation to a structured composition – originally referred to as structured process model (where each split has a corresponding join and all split-join combinations are properly nested) is desirable as described in detail in the next section.

Please note that VCL is used to describe a QoS-aware composition that has to be generated by the *Composition Service*. However, VRESCo does not enforce these VCL specifications if the QoS cannot be guaranteed at runtime. It is a best effort approach to find the services with the QoS as specified in VCL. However, if no service satisfies the VCL specification no valid composition can be generated. Therefore, VRESCo cannot enforce these specifications since most of the services that are composed are typically provided by other partners.

IV. QoS-Aware Composition and Optimization

The main goal of the composition and optimization approach is to assign an available service from the VRESCo registry to each feature in a VCL composition so that all global and local required constraints are satisfied and maximum number of optional constraints are fulfilled.

A. Composition Model

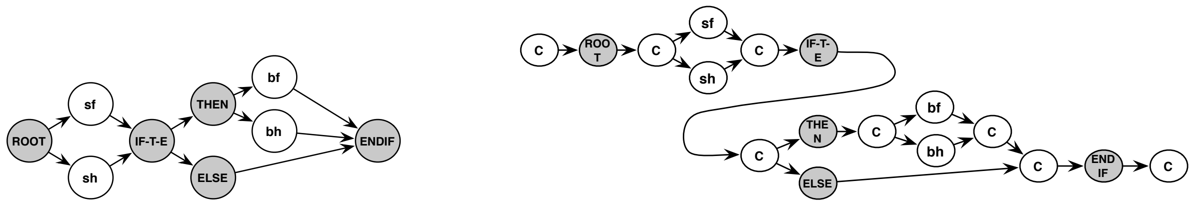
A VCL composition CS_{vcl} consists of a set of n features $F = \{f_1, f_2, \ldots, f_n\}$ to be composed. For each feature f_j, a set of m service candidates $S_j = \{s_{1j}, s_{2j}, \ldots, s_{mj}\}$ is available in VRESCo that implement a given feature. Each composition CS_{vcl} can be subject to global constraints $C_{gc} = \{I_{gc}, O_{gc}, P_{gc}, E_{gc}, Q_{gc}\}$. Each feature f_j can also have a set of constraints $C_{fc} = \{I_{fc}, O_{fc}, P_{fc}, E_{fc}, Q_{fc}\}$. These constraints represent a multi-set containing input constraints I, output constraint O, precondition constraints P, postcondition constraints E (effects), and QoS constraints Q. Constraints I, O, P, E specify restrictions on data of a feature or the composition itself and are not further considered in this paper because they do not play a major role during the QoS optimization process. The QoS constraints $Q_{gc} = Q_{fc} = (\langle q_{rt}, s\rangle, \langle q_l, s\rangle, \langle q_p, s\rangle, \langle q_{av}, s\rangle, \langle q_{ac}, s\rangle, \langle q_{tp}, s\rangle, \langle q_{rm}, s\rangle, \langle q_{sec}, s\rangle)$ represent a vector of labeled QoS constraints. The first element of each pair is the QoS attribute value (see the first two columns of Table I for details) and the second element $s \in H$ represents the constraint strength as defined in the hierarchy H. Additionally, each service candidate s_{ij} implementing a given feature f_j has a vector of QoS values (retrieved from the VRESCo registry). Based on [2], we merge the service candidates with their QoS attributes in a matrix $Q = (Q_{ij}; 0 \leq i < n; 0 \leq j \leq 7)$. Each row corresponds to a service candidate, whereas, the columns correspond to all the QoS values of a service candidate s_{ij}. We assume that the QoS attributes are numbered from 0 to 7 according to their order in Table I. In fact, our approach supports much more QoS attributes than the ones specified in Table I, however, for brevity we only choose a subset of values for this paper. A full list of available QoS attributes is given in [16].

B. Feature Resolution and Pre-filtering

Feature resolution is the process of querying and matching all services candidates that implement a given feature and its required feature constraints as specified in VCL. We assume that each feature of an application is defined in the VRESCo metadata model as part of the requirements engineering process. Additionally, we apply a pre-filtering technique to filter service candidates that do not fulfill feature QoS requirements with strength `required`. This reduces the number of service candidates for each feature that are later used in the optimization process, thus, speeding up the overall process. We use the VRESCo query language to implement the feature resolution.

C. Generating a Structured Composition

Existing composition engines (such as WS-BPEL) are capable of enacting structured process models and they usually cannot deal with unstructured process models (WS-BPEL has two exceptions: it allows cross links between parallel services and parallel blocks). Based on the approach presented by Eshuis et al. [7], we refer to a composition that is based on a structured process model as *structured composition*. As



(a) Annotated ADG (b) Structured Composition

Fig. 2. Generation of a Structured Composition

aforementioned in Section II, a structured model is desired and necessary for our approach based on the following reasons: i) it enables enactment of a structured composition on existing composition or workflow engines, thus, removing the need to implement an execution engine for VCL; ii) it allows to detect flaws in the unstructured composition such as deadlocks which may lead to runtime errors at a later stage; iii) it facilitates an efficient QoS aggregation based on well-known workflow and composition patterns [17]. QoS aggregation is needed during the optimization to determine an aggregation formula for a composition based on aggregation formulas the composition patterns [18]. The generation of a structured composition follows three basic steps: i) creation of an abstract dependency graph (ADG) to analyse the data flow of the business protocol as specified in VCL; ii) generation of the structured composition from the ADG and iii) annotating the control flow decisions in the structured composition with either AND (executed in parallel) or XOR (conditional execution).

Abstract Dependency Graph: Each feature in the VRESCo metadata model expects an input and an optional output message, that is composed of well-defined data concepts. A data concept is used to uniquely define a certain type from a domain model (such as a `Customer` or an `Address`). Additionally, atomic concepts exist to define atomic types such as integers or boolean. The abstract dependency graph can be defined based on the input/output data concepts of each feature. If a feature f_2 expects an input that is the output of a feature f_1, then we say f_2 depends on f_1. All data dependencies are captured in a graph data structure by using an adjacency list. The functions $input(f)$ and $output(f)$ in the following definition are used to get the input and output data concept of a feature. An ADG is given by the tuple (F, E):

- A set of features $F = \{f_1, f_2, \ldots, f_n\}$
- $E = \{(f, f') \in F \times F \,|\, output(f) \cap input(f')) \neq \emptyset\}$

In Figure 2(a) an ADG for the simple VCL from Listing 1 is depicted. In contrast to [7], we annotate the ADG with special nodes representing control flow information such as conditionals. The node IF-T-E is the abbreviation for IF-THEN-ELSE with its immediate successors THEN and ELSE representing the two possible conditional branches. Additionally, we add a root node that represents the composite service interface defined as global input constraint. In the ADG, this node has no inputs and outputs all data concepts of the composite service to its descendants. These special nodes are required in a later step to automatically annotate the incoming and outgoing edges of each feature to determine the correct branching type (AND, XOR). Additionally, we need this information to facilitate the generation of the executable composite service from the structured composition (not discussed in this paper). Loops, however, cannot be used in the algorithm from [7], because the graph has to be acyclic. Therefore, we insert a special loop start and end node and annotate all elements between the start and end node with their expected loop count given by the user to correctly calculate the QoS. During execution time, we refine the loop count using a simple analysis of the composition logs.

Generating the Structured Composition: This is done based on the ADG and the algorithm presented by Eshuis et al [7]. They provide a hierarchical view, representing a service (a feature in our terminology) as leaf node and blocks as non-leaf nodes. The graphical representation of the resulting structured composition is shown in Figure 2(b). The beginning and end of a block has a split and a join node (the node with C as label; C stand for composite). The beginning and end of a composition also have a composite node because the whole composition is also treated as a block. For a detailed description on how to construct that node, please consult the paper from Eshuis et al. After having generated the structured composition, we need to annotate each C-block with either AND or XOR to specify the branching type. Our rule is simple: we annotate each block with AND except blocks where the parent node is a special IF-THEN-ELSE node. In addition to that, we associate all services candidates S_j from the feature resolving process with each feature f_j in the structured composition graph. This information provides the input for generating the QoS-aware optimization problem.

D. QoS-Aware Optimization

The goal of the QoS-aware optimization is an optimal selection of one service candidate $s_{ij} \in S_j$ for each feature f_j where all the required global and local constraints are satisfied and a number of optional constraints from the constraint hierarchy H are satisfied. It is important to note that we do not search for the overall best solution in the search space, however, we search for the best solution within the boundaries given by the user constraints. We present two different approaches for modeling the QoS-aware optimization, a constraint optimization problem (COP) and an integer programming problem (IP). The reason for devising an IP solution as an alternative is based on the fact that most constraint-based approaches have scalability problems when applied to medium and large-scale practical optimization problems [19], however, the COP solution provides a simpler way of handling constraint hierarchies. In Table I we summarize all the QoS attributes including their aggregation formulas for the composition constructs supported in our approach. For the conditional (XOR) case, p_i is the probability that one path is chosen. For security and reliable messaging we assume that all services in the composition have to support the same security protocol and reliable messaging respectively to enable it for the composite service. In case of security, we only listed the aggregation for X.509 but it would be the same for every other mechanism. A more advanced mechanism will be considered in future work.

1) Constraint Optimization Problem: A COP is a constraint satisfaction problem (CSP) in which constraints are weighted and the goal is to find a solution maximizing a function of weighted constraints. A CSP is defined as a tuple $\langle X, D, C\rangle$ where X represents a set of variables, D_i represents the domain of each variable X_i and C represents a set of constraints over the variables X. A solution is an assignment of values from the variable's domain D to each variable $X_i \in X$ satisfying all the constraints C. For modeling our problem as a COP, we have to distinguish between *global constraints* and *feature constraints*. Both constraint types can have *required* and *optional* QoS constraints. Each required constraint has to be fulfilled, otherwise no solution can be found. All optional constraints (global and local) will be added to the objective function that has to be maximized. As aforementioned, all required feature constraints have already been pre-filtered, therefore, it is ensured that only service candidates have to be considered that fulfill all required feature constraints. This may reduce the number of constraints in the problem space.

a) Feature Constraints: Modeling feature constraints requires to add all service candidates S_j for each feature f_j as variables to the problem space. As we only want to select one service candidate from all available services S_j to execute a feature, we have to add the following *selection constraint* given that y_{ij} denotes the selection of a service candidate s_{ij} to execute a feature f_j (y_{ij} is modeled as a boolean decision variable[2]):

$$\sum_{i \in S_j} y_{ij} = 1, \forall j \in F \tag{1}$$

Each feature f_j can be subject to feature constraints, therefore, we need to add the following constraint for each feature constraint Q_{fc} to determine the selected QoS value q_{jk} of a feature (local selection). q_{jk} represents the selected QoS value for a given feature f_j.

$$q_{jk} = \sum_{i \in S_j} Q_{ik} * y_{ij}, \forall k \in Q_{fc} \tag{2}$$

Depending on the QoS attribute dimension (ascending, descending, exact) we need to add the corresponding constraints for each QoS attribute to capture whether an optional QoS constraint c_{jk} is satisfied (c_{jk} is represented as a boolean decision variable). The value q_{jk} is the value from constraint (2). For descending dimension, $c_{jk} = (q_{jk} \leq Q_{fc_k})$ is added, for ascending dimension $c_{jk} = (q_{jk} \geq Q_{fc_k})$, and for exact dimension the resulting constraint is $c_{jk} = (q_{jk} = Q_{fc_k})$.

Additionally, we use the following function (3) to map the constraint hierarchy levels to strength values that is then used to in the objective function. Please note that these values are flexible and can be changed to reflect a different mapping (e.g., give more weight to *strong* constraints).

$$strength(c) = \begin{cases} 20 & if\ c \in H_1 \\ 10 & if\ c \in H_2 \\ 5 & if\ c \in H_3 \\ 0 & otherwise \end{cases} \tag{3}$$

All the aforementioned constraints describe the selection of an optional feature QoS value. These constraints are added for each feature f_j and maximized as part of the objective function:

$$\max \sum_{j \in F} \sum_{k \in Q_{fc}} c_{jk} * strength(Q_{fc_k}) \tag{4}$$

b) Global Constraints: In order to add global constraints (required or optional ones), we first need to create an aggregation formula depending on the structured composition as shown previously in Figure 2(b) and the aggregation formulas as shown in Table I. We use a recursive algorithm to traverse the structured composition from the previous step and generate an aggregation formula for each feature f_j. For example, when aggregating the response time for the first composite block from Figure 2(b) (containing the feature `sf` and `sh`), the following aggregation constraint applies (k is the index for the QoS constraint, in this example it would be 0 for the response time):

$$a_k = \max_{j \in \{sf, sh\}} \{q_{jk}\} \tag{5}$$

In the following, we use a_k to represent the aggregation constraint of the k-th QoS attribute which is added for every global constraint that is specified by the user in the VCL specification. In case the global QoS constraint is required, we add another constraint depending on the QoS attribute dimension. For QoS with descending dimension, $a_k \leq Q_{gc_k}$ is added, for ascending dimension $a_k \geq Q_{gc_k}$, and for exact dimension the resulting constraint is $a_k = Q_{gc_k}$ represents the global QoS constraint where k is the QoS attribute index.

In case the global QoS constraint is optional, we have to add a decision constraint to check whether an optional constraint has been fulfilled. Again depending on the QoS attribute dimension, we add the following constraints: For descending dimension, $c_k = (a_k \leq Q_{gc_k})$ is added, for ascending dimension $c_k = (a_k \geq Q_{gc_k})$, and for exact dimension the resulting constraint is $c_k = (a_k = Q_{gc_k})$. Finally, we have to add these decision constraints multiplied with their strength value to the objective function from (4) to get the overall objective function:

[2]When using a boolean decision variable, 0 is used for false and 1 for true.

Attribute	Unit	Sequence	Conditional (XOR)	Parallel (AND)	Loop
Response Time (q_{rt})	msec	$\sum_{i=1}^{n} q_{rt}(f_i)$	$\sum_{i=1}^{n} p_i * q_{rt}(f_i)$	$\max\{f_1, .., f_n\}$	$q_{rt}(f) * c$
Latency (q_l)	msec	$\sum_{i=1}^{n} q_l(f_i)$	$\sum_{i=1}^{n} p_i * q_l(f_i)$	$\max\{f_1, .., f_n\}$	$q_l(f) * c$
Price (q_p)	per invocation	$\sum_{i=1}^{n} q_p(f_i)$	$\sum_{i=1}^{n} p_i * q_p(f_i)$	$\sum_{i=1}^{n} q_p(f_i)$	$q_p(f) * c$
Availability (q_{av})	percent	$\prod_{i=1}^{n} q_{av}(f_i)$	$\sum_{i=1}^{n} p_i * q_{av}(f_i)$	$\prod_{i=1}^{n} q_{av}(f_i)$	$q_{av}(f)^c$
Accuracy (q_{ac})	percent	$\prod_{i=1}^{n} q_{ac}(f_i)$	$\sum_{i=1}^{n} p_i * q_{ac}(f_i)$	$\prod_{i=1}^{n} q_{ac}(f_i)$	$q_{ac}(f)^c$
Throughput (q_{tp})	invocation/sec	$\min\{f_1, .., f_n\}$	$\sum_{i=1}^{n} p_i * q_{tp}(f_i)$	$\min\{f_1, .., f_n\}$	$q_{tp}(f)$
Reliable Messaging (q_{rm})	{true, false}	$q'_{rm} = \begin{cases} true & \forall_{1<i\leq n} q_{rm}(f_i) = true \\ false & \exists_{f_i \in F} q_{rm}(f_i) = false \end{cases}$			
Security (q_{sec})	{None,X.509,…}	$q'_{sec} = \begin{cases} X.509 & \forall_{1<i\leq n} q_{sec}(f_i) = X.509 \\ None & otherwise \end{cases}$			

TABLE I
QoS Attributes and Aggregation Formulas

$$\max\Bigg(\sum_{j\in F}\sum_{k\in Q_{fc}} c_{jk} * strength(Q_{fc_k}) + \sum_{k\in Q_{gc}} c_k * strength(Q_{gc_k})\Bigg) \quad (6)$$

The objective function (6) is then maximized by the solver to find an optimal solution within the constraint boundaries set by the user in the VCL description. All the values in our COP are scaled to integers by multiplying them with 100. Due to the fact that we only allow two decimal places in VCL we do not have any precision loss.

2) Integer Programming Approach: An IP optimizes a linear objective function that is subject to linear equality and linear inequality constraints. Compared to the CSP approach, there are a few changes that are needed when modeling the QoS-aware composition problem as IP. We have to define a new objective function calculating an overall utility value for each feature f_j considering the user's QoS constraints and their strength. Additionally, we need to linearize the aggregation rules for q_{ac} and q_{av} because they use the product to aggregate the QoS for a sequence and parallel execution of features.

a) Feature Constraints: Feature constraints are handled by using a utility function that is calculated for each service candidate. The selection constraint (1) is still valid in the IP formulation. For calculating the QoS utility function for each service, we first need to scale all the QoS values to a uniform representation. Contrary to other approaches in this area [2], we do not use simple-additive weighting scaling the values, however, we scale all values between [0, 100] depending on the percentage to which a QoS attribute of a service candidate fulfills the optional constraint imposed by the user. For example if the user specifies a optional availability constraint on a feature f_j with the value 0.95 and the QoS value of the service candidate is 0.99, we set the scaled value to 100 because the optional feature constraint is 100 percent satisfied (in fact is over-satisfied). The overall objective function is shown in (7):

$$max\Bigg(\sum_{j\in F}\sum_{i\in S_j} y_{ij} * \sum_{k\in Q_{fc}} scale(Q_{ik}, Q_{fc_k}) * strength(Q_{fc_k})\Bigg) \quad (7)$$

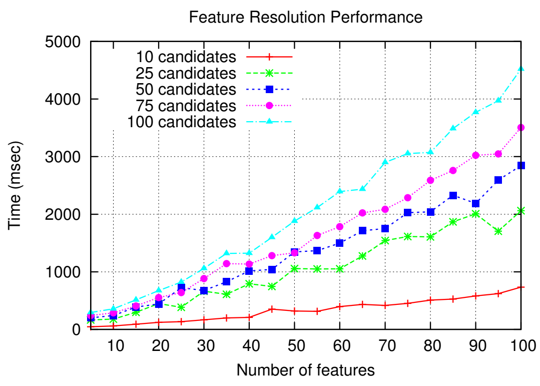
The function $scale$ scales the k-th QoS value Q_{ik} of a service candidate s_i between [0, 100] depending on the actual QoS feature constraint value Q_{fc_k} specified by the user in VCL and the QoS dimension (ascending, descending, exact). As a strength we use (3).

b) Global Constraints: For adding the global constraint, we follow a similar approach as in the CSP solution. We first aggregate the QoS attribute using a similar function as in the CSP approach, with the exception that we linearize the product aggregation rules using the ln (as shown in [2]). Whenever a global QoS constraint is required, we add a linear equality or inequality to the problem space. If a global constraint is optional, we add it to the overall objective function that has to be maximized.

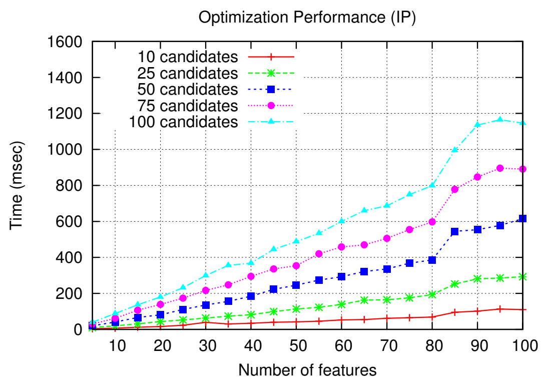
E. Implementation Aspects

The *Composition Service* was implemented as part of the VRESCo project in .NET/C# using the Windows Communication Foundation (WCF) for realizing the Web service communication. For querying the services as part of the feature resolution, we have used the VRESCo Query Language (VQL) to retrieve all deployed service instances. VCL was implemented by using the Microsoft Oslo framework[3], in particular using MGrammar, a toolkit for rapidly implementing domain-specific language for the .NET environment. The optimization algorithms are implemented by using the Microsoft Solver Foundation[4], a recently released optimization library supporting CSP, LP, MILP and Quadratic programming. For executing the generated composition we use the Microsoft Windows Workflow Foundation (WWF) [20]. The overall

[3]http://msdn.microsoft.com/en-us/oslo/
[4]http://code.msdn.microsoft.com/solverfoundation



(a) Feature Resolution Performance



(b) Optimization Performance (IP)

Fig. 3. Performance Evaluation

implementation of the composition service consists of approximately 5000 lines of C# code (without the VRESCo code itself).

V. Evaluation

In order to demonstrate the effectiveness of our approach we measure the performance of various crucial components such as the querying engine and the optimization approach. Therefore, we have written a tool to deploy features, categories, services and QoS values in VRESCo. Additionally, it generates VCL files with a given number of features and service candidates per feature. All the performance tests were executed on an Intel Xeon Dual CPU X5450 3.0 GHz with 32 GB of memory (although memory is not an issue in our tests).

A. Feature Resolution

The first important aspect is the performance of the feature resolution step. A query encodes all feature constraints (except optional QoS) to find concrete service candidates in the VRESCo registry as describe earlier in the paper. This step involves one query per feature in the composition using our VQL query language that are translated to SQL queries at runtime. We use a parallel query execution to speed up the performance as depicted in Figure 3(a) (the average of 10 repetitive runs). On the x-axis we represent the number of features in the composition (from 5–100). The y-axis shows the query time for depending on the number of candidates (10, 25, 50, 75, 100). For example, in a composition using 100 features, and each feature is implemented by 10 candidates the overall query time is less than a second (734 msec) and grows to more than 4 sec for 100 service candidates. Fortunately, the number of candidates is usually low (approx. 1–5), therefore, resulting in a very good overall query performance.

B. COP-based Optimization

For measuring the performance of the COP-based solver, we have generated small datasets as we expected a slow performance based on the complexity of the COP in general (NP-hard), and our problem in particular. In Table II the performance of the COP approach is depicted. The first column shows the number of features (from 5–25). Columns 2–4 show the performance in msec based on the number of candidates per feature (3, 5, 7). Each VCL file used for measuring the performance contains two global constraints and one local constraint per feature.

Features	**3 cand./feature**	**5 cand./feature**	**7 cand./feature**
	time in msec		
5	16	23	28
10	22	77	203
15	782	106952	exceeded
20	58789	exceeded	exceeded
25	exceeded	exceeded	exceeded

TABLE II
COP Performance

The results clearly show the bad performance (and negatively exceeded our expectations) as soon as the number of features is greater than 15 (and 5 candidates per feature). The value "exceeded" expresses that our timeout value of 120000 msec was exceeded. This makes the solver impractical for large QoS-aware optimization problems, however, for small scale problems it is still usable. Moreover, the current solver allows to save the internal solver state so that it can be reused and constraints can be added/changed or removed and then resolved. This will reduce the optimization time when the user specified constraints do not hold and the user has to relax some constraints.

C. IP-based Optimization

For measuring the performance of the IP solver, we have generated a number of VCL compositions (sequential and parallel constructs) with different number of features (from 5 to 100) and with increasing number of service candidates per feature (5, 25, 50, 75, 100). Each VCL file contains 4 global constraints and each feature has 4 feature constraints. We do not use any pre-filtering as part of the feature resolution in order to have the exact number of all the candidates per feature as given above. The performance results are depicted in Figure 3(b) (average of 10 repetitive runs). The x-axis shows the number of features, the y-axis shows the time in msec. Each function in the plot displays the runtime in msec based on the number of candidates per feature on the x-axis. The performance shows good results even for medium-size compositions (100 features), where a solution for 10 service candidates can be found in 110 msec and for 100 service candidates in 1145 msec.

D. End-to-End Performance

Finally, we present the end-to-end performance to compare all aspects involved to generate a composite service. In Figure 4, the performance of the VRESCo *Composition Service* is shown for an increasing number of features on the x-axis and 10 service candidates for each feature.

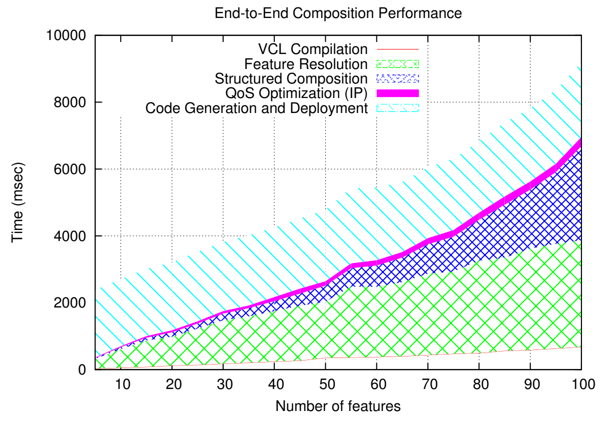


Fig. 4. End-to-End Performance for 10 Service Candidates

The performance of the different components is shown in different colors. Clearly, the feature resolution requires most of the time as shown in detail before. The code generation and deployment involves the generation of the WWF code and all required complex data types in form of C# classes (at least two for each feature having a complex input and output data type). This procedure is almost constant at around 2000 msec. The generation of the structured composition grows with the number of features. The other aspects are not significant in terms of their overall performance. The overall time needed to dynamically create and deploy a composition is reasonable but leaving some room for improving selected aspects.

VI. Discussion

The presented QoS-aware composition approach is based on the idea that compositions are hosted and can, therefore, be outsourced to realize Composition as a Service (CaaS). The approach is integrated into the VRESCo runtime environment and combines a number of novel concepts such as support for hard and soft QoS constraints and the automated generation of an executable composite service without the need for setting up and maintaining a composition infrastructure.

Currently, one of the main application areas for the CaaS approach is to specify QoS-aware composite services on a microflow level [6]. Compared to existing macroflow-based orchestration languages such as BPEL, these microflow services usually define a simple business protocol and are mainly used to define short running processes that do not have multi-party interactions. An efficient development and provisioning of QoS-aware microflows is important to implement larger, more coarse-grained business processes (macroflows). Therefore, the CaaS approach addresses adaptive behavior in a bottom-up manner by starting at the technical level of a business process implementation.

In particular, CaaS supports the development of adaptive behavior in two ways: Firstly, it leverages VRESCo as the implementation platform, therefore, we gain adaptiveness by using its dynamic binding and re-binding capabilities. In addition, by building upon VRESCo's feature-driven programming model, we additionally support adaptiveness by allowing to add new services or exchange existing ones transparently for higher-level macroflows (including runtime mediation). Secondly, adaptiveness is actively supported by the presented approach based on the fact that the QoS optimization can be re-executed for example by sending a request to the composite service or re-optimizing in an offline-mode. This ensures that the dynamically generated composite services always use the "best" service candidates that match all user constraints as initially specified in VCL.

VII. Related Work

Applying IP for solving the QoS-aware optimization problem has been used in other works. However, existing approaches only consider QoS as hard constraints, therefore, reducing the practical applicability and leading to over-constrained systems (i.e., systems with existing contradictory constraints in the problem space) that cannot be solved. As a consequence, we leverage constraint hierarchies [15] as a mechanism to solve such over-constrained systems by using labeled constraints to model hard and soft constraints. This approach increases the probability that a solution can be found and enables the specification of QoS that is "nice to have".

Zeng et al. [2] presented a solution of the composition problem by analyzing multiple execution paths of a composite service which are specified using UML statecharts. They model the composition problem using different approaches, including a local optimization approach and a global planning approach using IP. METEOR-S [21] is a comprehensive framework based on semantic Web technologies to specify, optimize and execute a composition. Their approach leverages WS-BPEL as an abstract process specification language where service templates are used to specify constraints on services and QoS that are matched against available services in a registry. The major difference of both aforementioned approaches is the fact that in our approach the user can specify global and local constraints using VCL, a DSL defined specifically for the purpose of specifying QoS constraints. Thus, we do not necessarily find the best solution overall, however, we search for the best solution within the boundaries specified by the user (which does not have to be the general optimum). We adapted some of the formalisms proposed by Zeng et al. to model our problem.

Guan et al. [22] are the first who propose a framework for QoS-guided service compositions which uses constraint hierarchies as a formalism for specifying QoS. Their idea of modeling functional requirements as hard constraints and using constraint hierarchies to model QoS is in line with the work presented in this paper, however, the authors use a branch and bound algorithm that is only capable of solving sequential compositions, whereas, our approach supports multiple composition constructs to be used. Additionally, the authors do not present any empirical evaluation to demonstrate the optimization performance of their approach.

Many other works related to QoS-aware optimization exist, for example a genetic algorithm based approach [3] or an approach based on a multi-objective stochastic program [5]. Common to these approaches is the fact that they only deal with the optimization problem itself, without presenting an end-to-end solution of the QoS-aware composition and optimization problem as we propose in this paper.

VIII. Conclusions

QoS-aware service composition remains a hot research area, since modern application architectures increasingly use services available from other business partners or from the Web. In this paper we have presented an approach for developing and optimizing QoS-aware composite applications by leveraging constraint hierarchies as a formalism to represent user constraints of different importance. The performance results of our approaches are promising that this approach is the foundation for QoS-aware runtime re-composition and re-optimization.

The main aspect of our future work is to leverage the VRESCo eventing infrastructure to perform efficient runtime re-composition to reduce the need for continuous querying for new or updated services or QoS. Additionally, we plan to develop and evaluate a set of other algorithms (such as genetic algorithms, tabu search and simulated annealing) to experiment with a larger number of features and service candidates.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

References

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

[2] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "QoS-Aware Middleware for Web Services Composition," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 311–327, May 2004.

[3] G.Canfora, M. D. Penta, R. Esposito, and M. L. Villani, "An Approach for QoS-aware Service Composition based on Genetic Algorithms," in *Proceedings of the Genetic and Computation Conference (GECCO'05), Washington DC, USA*. ACM Press, 2005.

[4] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient algorithms for web services selection with end-to-end qos constraints," *ACM Transactions on the Web*, vol. 1, no. 6, 2007.

[5] W. Wiesemann, R. Hochreiter, and D. Kuhn, "A Stochastic Programming Approach for QoS-Aware Service Composition," in *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'08), Lyon, France*, May 2008.

[6] U. Zdun, C. Hentrich, and S. Dustdar, "Modeling Process-Driven and Service-Oriented Architectures Using Patterns and Pattern Primitives," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 3, pp. 14:1–14:44, 2007.

[7] R. Eshuis, P. W. P. J. Grefen, and S. Till, "Structured service composition," in *Proceedings of the 4th International Conference on Business Process Management, Vienna, Austria*, 2006, pp. 97–112.

[8] F. Rosenberg, P. Leitner, A. Michlmayr, P. Celikovic, and S. Dustdar, "Towards Composition as a Service - A Quality of Service Driven Approach," in *Proceedings of the First IEEE Workshop on Information and Software as Service (WISS'09), co-located with the 25th International Conference on Data Engineering (ICDE'09), Shanghai, China*. IEEE Computer Society, Mar. 2009, pp. 1733–1740.

[9] R. Buyyaa, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, Dec. 2008.

[10] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "End-to-End Support for QoS-Aware Service Selection, Invocation and Mediation in VRESCo," Technical University Vienna, Tech. Rep. TUV-184-2009-03, Jun. 2009. [Online]. Available: http://www.infosys.tuwien.ac.at/Staff/rosenberg/papers/TUV-1841-2009-03.pdf

[11] F. Rosenberg, P. Leitner, A. Michlmayr, and S. Dustdar, "Integrated Metadata Support for Web Service Runtimes," in *Proceedings of the Middleware for Web Services Workshop (MWS'08), co-located with the 12th IEEE International Distributed Object Computing Conference (EDOC'08), Munich, Germany*. IEEE Computer Society, Sep. 2008.

[12] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "Advanced Event Processing and Notifications in Service Runtime Environments," in *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08), Rome, Italy*. ACM, 2008, pp. 115–125.

[13] P. Leitner, F. Rosenberg, and S. Dustdar, "Daios – Efficient Dynamic Web Service Invocation," *IEEE Internet Computing*, vol. 13, no. 3, pp. 72–80, May/June 2009.

[14] F. Rosenberg, C. Platzer, and S. Dustdar, "Bootstrapping Performance and Dependability Attributes of Web Services," in *Proceedings of the IEEE International Conference on Web Services (ICWS'06), Chicago, USA*, Sep. 2006.

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

[16] F. Rosenberg, "QoS-Aware Composition of Adaptive Service-Oriented Applications," Ph.D. dissertation, Technical University Vienna, Austria, Jun. 2009.

[17] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, , and A. Barros, "Workflow Patterns," *Distributed and Parallel Databases*, vol. 14, no. 3, pp. 5–51, Jul. 2003.

[18] M. C. Jaeger, G. Rojec-Goldmann, and G. Mühl, "QoS Aggregation for Service Composition using Workflow Patterns," in *Proceedings of the 8th International Enterprise Distributed Object Computing Conference (EDOC'04)*. IEEE CS Press, September 2004, pp. 149–159.

[19] K. M. Bayer, M. Michalowski, B. Y. Choueiry, and C. A. Knoblock, "Reformulating constraint satisfaction problems to improve scalability," in *Proceedings of the 7th Symposium on Abstraction, Reformulation and Approximation, Whistler, BC, Canada*, 2007, pp. 64–79.

[20] D. Shukla and B. Schmidt, *Essential Windows Workflow Foundation*, 1st ed. Addison-Wesley, 2006.

[21] R. Aggarwal, K. Verma, J. Miller, and W. Milnor, "Constraint Driven Web Service Composition in METEOR-S," in *Proceedings of IEEE International Conference on Services Computing (SCC'04), Shanghai, China*, Sep. 2004.

[22] Y. Guan, A. K. Ghose, and Z. Lu, "Using constraint hierarchies to support QoS-guided service composition," in *Proceedings of the IEEE International Conference on Web Services (ICWS'06), Chicago, IL, USA*. IEEE Computer Society, 2006, pp. 743–752.