Towards Composition as a Service – A Quality of Service Driven Approach

Florian Rosenberg, Philipp Leitner, Anton Michlmayr, Predrag Celikovic, Schahram Dustdar

Distributed Systems Group, Technical University Vienna
Argentinierstrasse 8/184-1, 1040 Vienna, Austria
{florian, leitner, anton, celikovic, dustdar}@infosys.tuwien.ac.at

***Abstract*— Software as a Service (SaaS) and the possibility to compose Web services provisioned over the Internet are important assets for a service-oriented architecture (SOA). However, the complexity and time for developing and provisioning a composite service is very high and it is generally an error-prone task. In this paper we address these issues by describing a semi-automated "Composition as a Service" (CAAS) approach combined with a domain-specific language called VCL (Vienna Composition Language). The proposed approach facilitates rapid development and provisioning of composite services by specifying what to compose in a constraint-hierarchy based way using VCL. Invoking the composition service triggers the composition process and upon success the newly composed service is immediately deployed and available. This solution requires no client-side composition infrastructure because it is transparently encapsulated in the CAAS infrastructure.**

I. INTRODUCTION

Over the last years, the Software as a Service (SaaS) concept has gained momentum as a means to offer software as well-defined services over the Internet. Implementing SaaS is typically achieved by leveraging service-oriented architecture (SOA) design principles such as loose coupling and self-describing service interfaces. Web services as one technology for implementing SOAs can help to achieve this vision by providing standards such as WSDL [1] as service description language and SOAP [2] as transport protocol.

One of the main benefits of implementing a well-shaped SOA is the ability to compose new functionality out of existing services into so-called "composite services", thus significantly increasing reuseability of existing services. This process itself is called composition [3] or orchestration and needs a composition engine to enact a composed service (or application)[1].

Typically, two major issues have to be addressed before adopting service compositions: Firstly, which composition approach should be used allowing an adequate description of functional and non-functional aspects of a composed service? These include functional aspects such as the services to be used, non-functional aspects such as Quality of Service (QoS) concerns as well as the specification of the control and/or data flow (depending on the composition language). Secondly, how are these composed services deployed and provisioned to allow other software artifacts to invoke them?

[1]We use the terms composite service, composite application and composition interchangeably in this paper.

Currently, several approaches to engineer composite applications exist. The vast majority of developers use a purely static approach to build a composition by manually selecting available services and specifying the control flow to define the composition logic. Static compositions can be built for example by using WS-BPEL [4] or the Microsoft Windows Workflow Foundation [5]. The static nature of a composition poses a real problem, for example, when services need to be dynamically selected or exchanged from a pool of similar services based on changing Quality of Service (QoS) attributes such as response time, throughput or availability. So-called dynamic service adaptation mechanisms for static approaches have been introduced to overcome this problem and increase availability and fault-tolerance of Web service compositions [6], [7].

In terms of service deployment and provisioning, a service composition infrastructure is required to enable the deployment of composite services and atomic services (if not hosted otherwise). Currently, most deployment aspects include a number of (often 'hand-crafted') tasks leading to a successful deployment. This process is usually very cumbersome and error-prone. Especially for small and medium sized enterprises (or even individuals) the required infrastructure to host composite services is often not available or simply too expensive in terms of acquisition cost and maintenance.

In this paper we address these two aforementioned issues by adding the CAAS approach to our VRESCO SOA runtime [8], [9], [10], [11], [12] and enable the specification of composite applications with VCL. VRESCO itself provides a uniform programming model and SOA runtime that addresses issues such as service publishing, querying, versioning and service mediation based on a uniform service metadata model. A detailed description of all the VRESCO aspects is out of scope of this paper, we refer the interested reader to our previous work.

The contributions of this paper can be summarized as follows: Firstly, we put forward a textual domain-specific language (DSL) named VCL (Vienna Composition Language) that was developed solely for the purpose of specifying service compositions. The novelty of VCL is the ability to specify constraints describing functional and non-functional aspects of a composition using a constraint hierarchy-based approach. Quality of service (QoS) is a first-class entity of the DSL enabling a QoS-aware composition.

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.153

Secondly, we bridge the gap between composite service specification and provisioning by enabling "Composition as a Service" (CaaS). To this end, we leverage VRESCo as an infrastructure (including metadata for the services stored in a service registry) to enhance it with a composition environment to receive and process composition requests (written in VCL). During runtime, a composition request is transformed into a constraint optimization problem to find an optimal selection of services that match the user's constraints. However, in this paper we focus on the end-to-end system rather than on the QoS-aware optimization algorithms.

This paper is organized as follows: Section II presents some of the related work. Section III and IV describe the approach and rationale behind CaaS and VCL. Some implementation aspects are highlighted in Section V. Section VI discusses some of the issues and limitations related to this work and Section VII concludes this work and highlights some future work.

II. Related Work

Applying DSLs for composing service-oriented systems is a relatively new idea, whereas DSLs have been successfully applied in other areas (also within the SOA domain). Oberortner et al. [13] discuss the use of DSLs for SOAs in general, with a special focus on model-driven development and process-driven SOAs [14]. The authors differentiate between DSLs for domain experts (high-level DSLs) and DSLs for technical experts (low-level DSL). According to their classification, VCL can neither be explicitly classified as high-level DSL nor as low-level DSL because it abstracts from low-level semantic and syntactic issues (such as constructs in WS-BPEL or any other composition language) but requires some technical understanding about the domain of "service composition" to be practically usable.

In [15], the author describes the WebDSL approach, a domain-specific language for dynamic Web applications. It allows the specification of domain models, presentation logic, page flows and access control [16]. Similar to our approach, WebDSL abstracts from the complexity of the underlying execution languages and runtimes (JSF, Hibernate and Seam in their approach). Besides abstracting from the underlying runtime, in our approach we additionally introduce an optimization layer in between the language specification and the generation of the executable composition to optimize the local and global QoS constraints that have been specified by the user.

JOpera [17], developed at the ETH Zürich, provides a visual composition language that focuses on an interactive environment allowing users to visually specify, design and test their compositions. Their approach does not focus on pure SOAP-based services, it also handles arbitrary Java or Enterprise JavaBeans and RESTful services. In contrast to their approach, we go into a different direction by providing a semi-automated approach using a textual DSL that gives developers a simple language to rapidly developing and deploying a composition without requiring any composition infrastructure.

In terms of QoS-aware optimization of compositions, Zeng et al. [18] provided some of the fundamentals in the Web service community. They use the well-known Integer Programming (IP) technique to find an optimized composition in terms of QoS that satisfies all the global and local QoS constraints. This is based on the assumption that for each abstract service in a composite service a pool of candidate services exists. The main challenge is the encoding of various QoS constraints and the aggregation of QoS values in the workflow according to the execution path in a composite service. Their approach can be seen complementary to our work, however, our overall approach is not focused on one specific formalism for QoS-aware optimization. VCL itself allows the specification of given QoS requirements, either locally on given services as well as globally. Our work is also capable of integrating different optimization algorithms, such as approaches based on stochastic models [19], [20].

In [21], the authors discuss an approach for interleaving planning and execution of service compositions by means of a special language called XSRL (XML Service Request Language). It enables users to specify goals and constraints for a (pre-compiled) composite service where the services are dynamically bound in the composition, e.g., by using UDDI (Universal Description Discovery and Integration). Their approach is based on AI planning and constraint satisfaction techniques to fulfill a service request. In contrast, we focus on providing a language and runtime to create and deploy a composite service that has various constraints in terms of functionality and QoS. Once a composite service is deployed our approach is still able to re-bind to another service once the QoS changes since the initial deployment of the composition.

III. Composition Approach

The Composition as a Service (CaaS) approach is based on the idea of reducing the complexity involved when developing a composite application, e.g., such as with WS-BPEL. Typically, the composition process can be divided into *service discovery*, *composite service specification*, *deployment*, *testing* and *runtime monitoring* (assuming that the requirements of the composite service in terms of its functionality have already been defined). The CaaS approach in combination with the VRESCo SOA runtime mainly encompasses all these steps (except testing) in a holistic approach. Service discovery is not done by querying public registries but VRESCo provides a mechanism to publish services in its own registry and associate them with expressive metadata which can later be queried by a powerful query language.

A. CaaS Overview

An overview of the CaaS approach is depicted in Figure 1. From an end-user perspective the developer (on the client-side) specifies the composition in VCL and uses the client library to invoke the composition service at the VRESCo runtime. The client library provides a convenient way to access the VRESCo core services (such as publishing, metadata, querying, etc). Additionally, the client library compiles the VCL

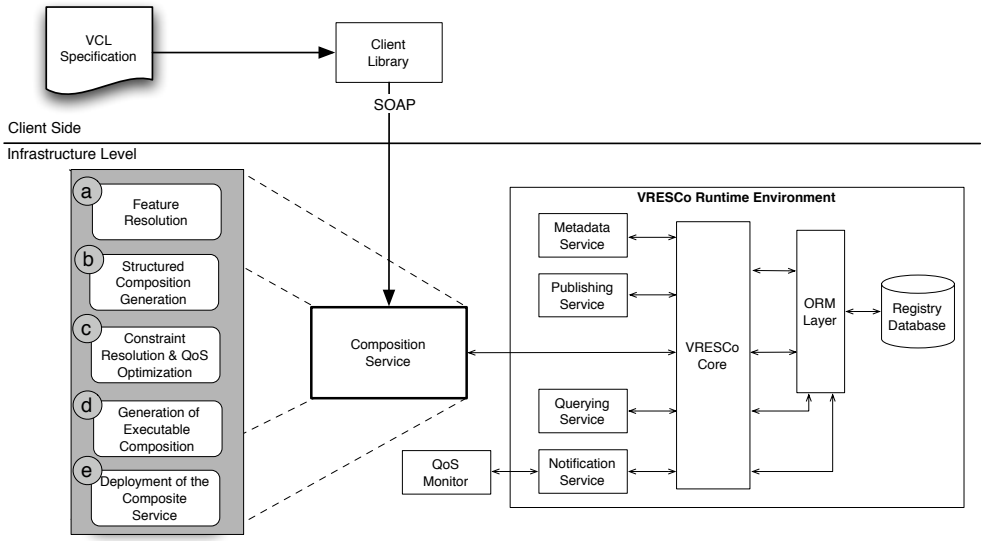

Fig. 1. Architectural Overview of Composition as a Service with VCL

specification and checks for static errors to avoid contacting the composition service using invalid input. Once a statically correct VCL specification is sent to the VRESCO runtime, the five steps (a) to (e) in the grey box on the left side have to be executed to successfully deploy and provision a composite service. Step (a) comprises the resolution of features (service operations are denoted as features in our model – see below) that are required for the composition. Resolving all features implies a translation of the feature requirements into a query that is offered by the VRESCO querying service. In step (b) a data flow analysis of the VCL specification is performed to determine dependencies among the service and to generate a fully structured composition [22]. Once all features and data dependencies have been resolved, step (c) is activated where all the constraints in the VCL specification have to be satisfied and the QoS optimization problem defined by the QoS constraints has to be resolved. A non-satisfiable constraint leads to an error and will send a notification back to the user to allow changes, e.g., by relaxing some constraints. Assuming all constraints are satisfied, the actual generation of the executable composite service happens in step (d). Step (e) finalizes the process by deploying the generated composite service and sending the newly deployed service endpoint back to the user. Additionally, the new service is registered in the VRESCO registry using the publishing service.

B. Service Model

Before going into the details of VCL, we need to introduce the foundations of our service model as it is essential for VCL. The user needs to be able to define what should be composed and the composition engine needs be able to retrieve metadata about the services to match its capabilities with the user's requirements. In our approach, we define a service by its *functional* and *non-functional characteristics* [11].

1) Functional Characteristics: The functionality of a service is described by the VRESCO metadata model, an abstract, feature-driven model for defining what functionality is offered by a service. In Figure 2, we have depicted our basic metadata model for modeling services, their categories, features, pre- and postconditions. In this model we have to abstract from the technical service implementation to achieve a common understanding what a service does and what it expects and provides. In a typical SOA environment, there may be multiple services that facilitate the same business goal, therefore, we also need a way to group services according to their functionality. In the following, we use *italic* font to represent model elements and `typewriter` to indicate instances of a model element.

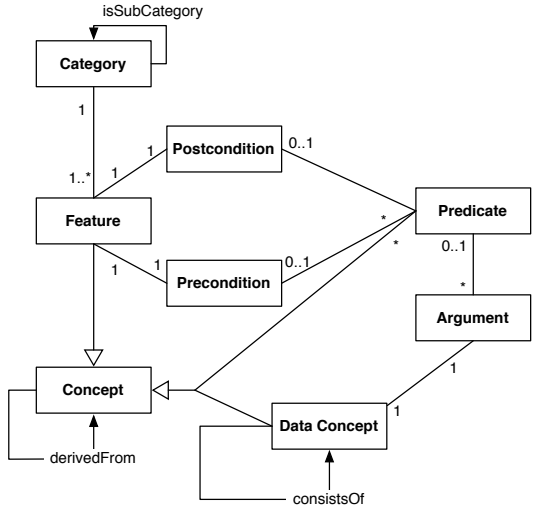

Fig. 2. VRESCo Metadata Model

The main building blocks of the VRESCo metadata model are *Concept*s. A *Concept* is the definition of an entity in the domain model (e.g., a generalized model element `Person` and a special variant of it called `Customer`). We distinguish between three different types of *Concept*s:

- *Feature*s represent activities in the domain that perform a concrete action, e.g., `Calculate_Loan`, `Create_Customer`.
- *Data Concept*s represent concrete entities in the domain (e.g., customers, addresses or bills) which are defined using other *Data Concept*s (e.g., the concept `Customer` might consist of `Customer_Id`, `Customer_Name`, and `Address`) and/or atomic elements such as strings or numbers.
- *Predicate*s represent domain-specific statements that either return `true` or `false`. Each *Predicate* can have a number of *Argument*s that express their inputs. For example, a (state) predicate for a *Feature* `Create_Customer` could be `Customer_Not_Exists`, expressing that a customer must not exist before invoking that feature.

*Concept*s have a well-defined meaning specific to a certain domain. Concepts may be derived from other concepts; that is specifically interesting for *Data Concept*s, e.g., it is possible to define the concept `Premium_Customer` which is a special variant of the more general concept `Customer`.

Each *Feature* is associated with one *Category* expressing the purpose of a service (e.g., `Customer_Relationship_Management`). Each category can have additional subcategories to allow a more fine-grained differentiation. Each *Feature* has a *Precondition* and a *Postcondition* expressing logical statements that have to hold before and after the execution of a *Feature*. Both types of conditions are composed of multiple *Predicate*s, each having a number of (optional) *Argument*s that refer to a *Concept* in the domain model (indirectly through a *Data Concept*). A *Predicate* can be used to express data flow or observable state related information that may be necessary when composing a service. For example, a *Feature* called `Create_Customer` could have a precondition `Customer_Not_Registered(Customer)` expressing that a customer must not be registered before invoking it. In general, predicates can be specified by the developer to explicitly define flow and state behavior, however, they are not required or enforced by the service implementation upon execution time. This kind of metadata only provides knowledge which is required later, when performing semi-automated or fully automated service composition, where such pre- and postconditions are a required means to guide the composition process for stateful services.

The VRESCo client library provides an API to define the aforementioned metadata programmatically. When concrete service instances are published, the mapping between a feature and a concrete service operation can be defined. Additionally, the input and output concepts of a feature can be mapped to the input and output of the concrete service instances. Having multiple services which implement the same feature enables a dynamic binding and dynamic interface mediation between similar services [23]. This powerful concept enables a transparent handling of service invocations based on features and is foundational for the approach described in this paper.

2) Non-functional Characteristics: Besides the functional characteristics, a set of QoS attributes is associated with every service. QoS is a useful measure to distinguish well performing from bad performing services. In general, one has to distinguish between deterministic and non-deterministic QoS. The former encompasses values which are known when a service is invoked whereas the latter are unknown at invocation time, therefore, they have to be gathered through runtime observation. In [24], we have described an approach for measuring QoS attributes from a client side perspective that we use within the VRESCo environment (see the QoS Monitor in Figure 1).

Attribute	Formula	Unit
Response Time	$q_{rt}(n) = \frac{1}{n}\sum_{i=0}^{n} q_{rt_i}$	msec
Latency	$q_{la}(n) = \frac{1}{n}\sum_{i=0}^{n} q_{la_i}$	msec
Availability	$q_{av}(t_0, t_1, t_d) = 1 - \frac{t_d}{t_1 - t_0}$	percent
Accuracy	$q_{ac}(r_f, r_t) = 1 - \frac{r_f}{r_t}$	percent
Throughput	$q_{tp}(t_0, t_1, r) = 1 - \frac{r}{t_1 - t_0}$	invocation/sec
Price	n/a	per invocation
Reliable Messaging	n/a	{true, false}
Security	n/a	{None,X.509,...}

TABLE I
AVAILABLE QOS ATTRIBUTES

All QoS attributes considered in our approach are shown in Table I. For each attribute we list a distinct name, a formula how the attribute is calculated in case of non-deterministic attributes or "n/a" if it is deterministic (such as price, reliable messaging and security). The response time $q_{rt}(n)$ is the duration that a service request needs on the wire plus the execution time of service at the service provider. It is calculated as the average value of n individual measuring points q_{rt_i}. The latency $q_{la}(n)$ is the time a request needs on the wire and the average is calculated the same way as the response time. The availability $q_{av}(t_0, t_1, t_d)$ is the probability a service is up and running and producing correct results; (t_0, t_1 are timestamps, t_d is the time the service was down). The accuracy $q_{ac}(r_f, r_t)$ is the probability of a service to produce correct results where r_f denotes the number of failed request and r_t denotes the total number of requests. The throughput $q_{tp}(t_0, t_1, r)$ is the maximum number of requests a service can process within a certain time period (denote as $t_1 - t_0$), and r is the total number of requests during that time.

QoS attributes are automatically available within the

VRESCo runtime and are monitored using our approach from [24]. The observed values are automatically associated with the corresponding service operations in the VRESCo registry and can be queried using the querying service (see Figure 1).

IV. Vienna Composition Language

The main goal of VCL is to provide an intuitive and simple DSL for the purpose of composition within the VRESCo environment. It enables to capture what a composition should do and what QoS is required from a global perspective and also what QoS is required from individual services in the composition (local perspective). Additionally, constraints on individual services can be imposed, such as on inputs and outputs of each service, preconditions or postconditions that have to be fulfilled. A main concept in VCL is the fact that we group QoS constraints into constraint hierarchies to address the problem that not all QoS attributes can always be fully satisfied and not necessarily have to be. Consider an example, where a user specifies two global constraints on a composition expressing that the response time should be $\leq 5000msec$ and the availability should be ≥ 0.95. Unfortunately, the composition system cannot fulfill the availability constraint because its actual value is 0.935, thus, the composition process fails due to a violating QoS constraint. By using constraint hierarchies one can add a strength to QoS attributes to express its importance in a hierarchical way. Traditional constraint-based approaches usually fail when a constraint is violated and no solution exists that fulfills all constraints. Such systems of constraints are called over-constrained systems. Constraint hierarchies [25] have been proposed to solve such system by associating a strength or preference value with each constraint expressing its importance in the constrain resolution. In VCL we use four different hierarchy values per default: {`required`, `strong`, `medium`, `weak`} can be specified for each QoS global or local QoS constraint and will be respected by the composition algorithm.

The service model presented in Section III is an integral part for the overall CaaS approach, as VCL has to rely on a strong runtime support that is able to generate an executable composite service from that abstract description specified in VCL. The core elements of a VCL specification are visualized in Figure 3 and described in detail in the subsequent sections.

A VCL specification of a composite service CS is a tuple $CS =< FD, FC, GC, BPS >$ with the following elements: FD represents the *feature definitions* that specify which features will be composed. Each feature has an associated *category* and an *invocation type* defining whether a feature has to be invoked synchronously (type `sync`) or asynchronously (type `async`). The definition of features and categories follows the notation we introduced in Section III. Due to the fact that categories can have multiple subcategories (as denoted in Figure 2), the specification of categories allows a wildcard character $\star$ to refer to a specific category within the category tree without specifying the whole path in the category tree.

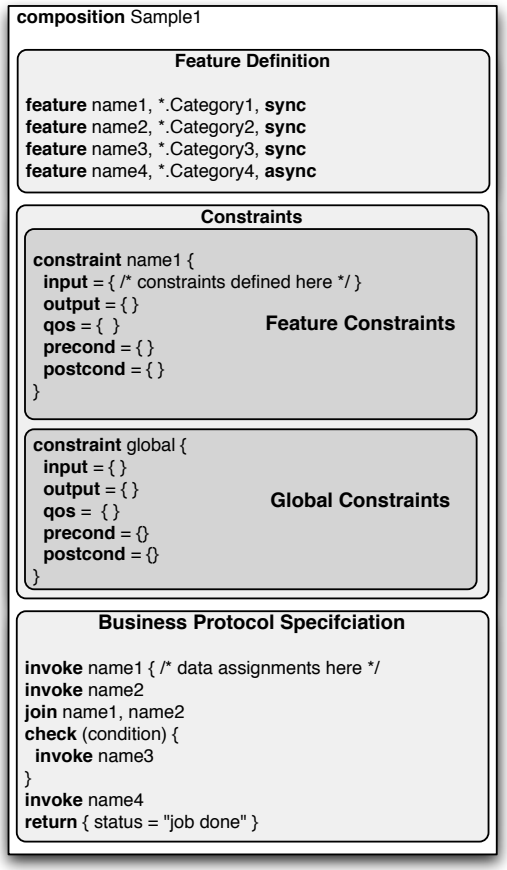

Fig. 3. VCL General Schema

FC represents a set of (optional) *feature constraints* that can constrain the input, output, QoS, precondition and postcondition of each feature. The input and output constraints restrict the data which is required as an input or output of a feature. Preconditions and postconditions can be used to express assertions that need to be valid before or after the execution of a feature, respectively. QoS constraints can be associated with a strength to express its preference ("required" strength is the default).

GC represents the *global constraints* for the overall composition. Similar to feature constraints, global constraints can be used to restrict input, output, QoS and specify preconditions and postcondition for the resulting composite service CS. The semantics of input and output is used to define input and the output data of CS. In other words they define the input and output of the composed service interface. Again, QoS constraints can be associated with a strength value.

BPS represents an abstract *business protocol specification* that defines what service should be invoked and can also specify simple conditional execution or loops. The business protocol specification does not have to specify a complete composition, as it has to be done in WS-BPEL. Many aspects, such as determining the execution order, are done automati-

cally by the *composition service* that acts as the backend to VCL. For example in Figure 3 we have several statements illustrating various aspects of a control flow definition, e.g., the invocation of the given feature using `invoke` (possible data assignments are not shown in the figure). They can either be executed sequentially or in parallel, depending on the data flow. If, for example, feature "name2" has no dependencies on "name1" they both can be executed in parallel. VCL is flexible in terms of the concrete BPS formalism. Currently we use a graph-based approach to specify the BPS, however, we are not restricted to it. It is also possible to use a label-transition system to specify the business protocol of a composition. Currently we support the most important control flow constructs such as loops, conditional, splits, joins, etc.

A. VCL *by Example - A RestaurantGuide Service*

We depict a simple example composing a restaurant guide service which can be used for example on mobile devices to quickly locate and reserve a restaurant for the same day at a specific time, for a number of persons. The VCL specification is shown in Listing 1. We assume that all the services that are used have been published previously in the VRESCO registry and associated with metadata. The QoS monitor is aware of the services and provides up-to-date quality information. The composition uses four features (line 3-6) defined in the VRESCO registry, and a several services are associated with them (e.g., some restaurant services, local taxi services and map services such as Google Maps, Yahoo, etc). Those services are mapped to features using the VRESCO client library. It is important to note that feature `sendMapInfo` (line 6) is invoked asynchronously and sends the map information directly per email to the invoker of the composite service.

After the feature definition, a number of constraints on the individual features can be defined. In our case, we just define one QoS constraint for the feature `sendMapInfo` to restrict its response time to $\leq$ 5000msec to ensure that we receive the map information about the location (per email) within less then five seconds (line 8–12).

The global constraints for the whole composition are described from line 15 to 33. They define the external service interface of the composed service using the input and output constraint. The QoS constraint defines some general QoS properties that have to be fulfilled – depending on their hierarchy level – before the composition can be generated and deployed. In this example we only define the response time as "required" constraint. All the other ones are "nice to have" with a preference on price over security.

From line 35 to 62, the business protocol is specified. In our current version of VCL we use a procedural way to specify the composition by having a number of control flow and synchronization constructs. Data flow is also supported by allowing to assign and read data from a feature. For example a complex data structure can be accessed and initialized with new values as shown from line 36 to 39. The `return` statement terminates the composite service execution by returning the required data as specified in the global output constraint.

```
1  composition RestaurantGuide
2
3  feature findRestaurant, *.Restaurant, type sync
4  feature reserveRestaurant, *.Restaurant, type sync
5  feature orderTaxi, *.Transportation, type sync
6  feature sendMapInfo, *.LocationServices, type async
7
8  constraint sendMapInfo {
9    qos = {
10     responseTime <= 5000, required;
11   }
12 }
13 # other features constraints cut for readability
14
15 constraint global {
16   input = { # inputs of the composite service interface
17     string restaurantType;
18     string restaurantQuality;
19     string name;
20     string phoneNr;
21     string emailAddress;
22     string nrOfPersons;
23     string time;
24   }
25   output = { # output of the composite service interface
26     boolean reservationStatus;
27   }
28   qos =  {
29     responseTime <= 5000, required;
30     price < 5, strong;
31     security = X.509, weak;
32   }
33 }
34
35 invoke findRestaurant {
36   RestaurantRequest [
37     food = restaurantType;
38     stars = restaurantQuality;
39   ]
40 }
41 # some restaurants found?
42 check (findRestaurant.RestaurantResponse.hits > 0) {
43   invoke reserveRestaurant {
44     ReservationRequest [
45       name = name;
46       phone = phoneNr;
47       email = emailAddress;
48       time = time;
49       count = nrOfPersons;
50     ]
51   }
52   invoke orderTaxi { // cut for readability
53   }
54   invoke sendMapInfo { // cut for readability
55   }
56 }
57 return {
58   reservationStatus =
59     findRestaurant.RestaurantResponse.hits > 0 &
60     reserveRestaurant.status &
61     orderTaxi.reservationStatus;
62 }
```

Listing 1. RestaurantGuide Composition

B. Composite Service Generation

Up to now, we have mainly focused on VCL to specify a QoS-aware composite service. Due to space restriction we can only briefly go over all the required steps, without going into too much detail. When sending the VCL composition request to the composition service (see Figure 1), we can already assume that the VCL input is valid has no static errors because it has been validated using the client library. It includes a compiler for the DSL that transforms the VCL input into a C#-based object model. This object model is the main input for all the other processing step that follow.

a) Feature Resolution: Feature resolution is the process of querying all services candidates that implement a given feature and its constraints as specified in VCL. We assume that each feature of a business application is defined in the VRESCO metadata model as part of the requirements engineering process. Each feature is traversed by the feature resolution algorithm and translated into a VQL (Vienna Query Language) query, VRESCO's query language. For example, the feature `findRestaurant` from Listing 2 will result in the following VQL query:

```
1 var fquery = new VQuery(typeof(ServiceRevision));
2 fquery.Add(Expression.Eq("Operations.Feature.Name",
3                          "findRestaurant"));
4 IList<ServiceRevision> features =
5     querier.FindByQuery(fquery, QueryMode.Exact)
6     as IList<Feature>;
```

Listing 2. Feature to Query Translation Composition

In addition to querying for the feature name, we also add all *required* QoS constraints, input, output, pre- and postconditions as criteria to the query to ensure that a feature fulfills all its constraints. This is achieved by adding a criterion for each constraint to the query in Listing 2 (similar to the criterion in line 2–3). After the feature resolution process, we have resolved all features and have a set of services for each feature that fulfill all required *feature constraints* (a list of `ServiceRevision` instances internally in VRESCO).

b) Structure Composition Generation: In this step – it happens in parallel to the feature resolution as there are no dependencies – the composition service analyzes the business protocol specification of the VCL specification. The business protocol does not necessarily be complete in the sense that the user has to specify the service execution order exactly. However, in our example in Listing 1 we specify it completely to illustrate most of VCL's features. The main goal of this step is to analyze data dependencies among the service invocations to generate a structured service composition (i.e., determine the correct execution order). We have adopted and extended the approach described by Eshuis et al. [22] to generate a structured composition and also to calculate the overall QoS of a composition.

c) Constraint Resolution and QoS Optimization: In this step, we formulate a constraint satisfaction problem (CSP) which has all the service candidates (grouped per feature) from the previous feature resolution as an input. All the local (only optional ones) and global constraint (required and optional ones) from the VCL specification will be added as constraints. We use the structured composition graph from the previous step to aggregate all the global QoS constraints based on the control flow of the composition and their aggregation formulas. The CSP can have multiple solutions, thus, we associate a weight with different constraint hierarchies levels to find the optimal solution.

d) Generation of Executable Composition: In this step we transform the structured workflow and the information which concrete service to invoke (from the constraint resolution process) into a Windows Workflow Foundation (WWF) representation. This is achieved by traversing the structured composition model and generating the respective workflow activities in XAML (Extensible Application Markup Language).

e) Deployment of Composite Service: The deployment step is the simplest step and just involves parsing the previously generated workflow and hosting it using the WWF provided `WorkflowServiceHost` class. Finally, the endpoint of the composite service that is implemented by the generated workflow is returned to the caller, and the composite service is ready to accept client requests.

V. IMPLEMENTATION

VCL is implemented as a DSL by using the Microsoft Oslo Toolkit[2]. Oslo assists in defining the DSL grammar and provides a compiler to translate the DSL into a custom object model. This object model contains all our information encoded in that DSL. The composition service that does all the DSL processing described earlier in the paper, is implemented as a core service within the VRESCO platform. VRESCO itself is also implemented using the Microsoft .NET framework (C# in particular) and uses the Windows Communication Foundation (WCF) as a technology for implementing Web services. The CSP is solved using NSolver[3]. All VRESCO core services, including the composition service can be accessed using a pre-built client library – available for Java and .NET – implementing an object-oriented connector for VRESCO to enable a better productivity while using VRESCO.

VI. DISCUSSION

VCL is not a generic language that is able to solve all QoS-aware service composition issues. The syntax of the language is flexible in the sense that new language constructs or QoS attributes can be added quickly. Nevertheless, the semantics behind the language is targeted specifically to the VRESCO environment and its metadata model [11]. The ability to manage services in VRESCO and associate them with metadata (e.g., features that map to service operations), and to dynamically query and invoke services are core features leveraged by the composition service.

Another important aspect to consider is the granularity of the composite service that can be implemented with VCL. By design, VCL was developed with a special focus on QoS-aware service compositions and is not targeted to solve large-scale multi-party workflows with several interactions (possibly involving human actors). For such tasks, it is favorable to use a well-know workflow engine. Besides, the right granularity, we claim that the composition service can generate an executable composite service according to VCL specification. This is true, however, there are still cases where an executable workflow cannot be generated, for example, due to an insufficient specification, or the lack of expressiveness in VCL (e.g., missing functionality). In such cases, VCL still has some value by generating a composite service template that can then be refined and deployed manually.

[2] `http://msdn.microsoft.com/oslo`
[3] `http://www.cs.cityu.edu.hk/~hwchun/nsolver/`

Based on the current VCL implementation, it is important to note that the way how the business protocol for a composite service is defined is not fixed. We currently use a graph-based model, similar to existing languages such as WS-BPEL, to define the invocation order and the control flow. However, the DSL implementation is very flexible in terms of adding new language constructs to support other formalisms to specify the composition logic. While the business protocol specification might change over time, the rest of the language is stable

VII. CONCLUSIONS

In this paper we have illustrated the concept of "Composition as a Service" which aims at reducing the need for a composition infrastructure and allows to compose and deploy services on the fly. We contribute a DSL called VCL as the core language for using the CaaS approach. VCL is a QoS-driven composite service specification language following a constraint-hierarchy based approach to specify what services are needed and what QoS are required and desired.

We are currently setting up a testbed with thousands of services that all simulate different QoS values. This will allow us to thoroughly evaluate and tune our constraint resolution algorithms to demonstrate an efficient end-to-end composition. In addition, we plan to incorporate dynamic reconfiguration of a composite service if local and global QoS constraints specified at design time no longer hold during runtime (based on changing QoS of service that implement a specific feature).

ACKNOWLEDGMENT

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] *Web Services Description Language (WSDL) 1.1*, World Wide Web Consortium (W3C), 2001. [Online]. Available: http://www.w3.org/TR/wsdl

[2] *SOAP Version 1.2*, World Wide Web Consortium (W3C), 2003. [Online]. Available: http://www.w3.org/TR/soap

[3] S. Dustdar and W. Schreiner, "A Survey on Web services Composition," *International Journal of Web and Grid Services*, vol. 1, no. 1, pp. 1–30, 2005.

[4] *Web Service Business Process Execution Language 2.0*, OASIS, 2006. [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

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

[6] O. Moser, F. Rosenberg, and S. Dustdar, "Non-Intrusive Monitoring and Adaption for WS-BPEL," in *Proceedings of the 17th International International World Wide Web Conference (WWW'08), Beijing, China*, Apr. 2008.

[7] O. Ezenwoye and S. M. Sadjadi, "RobustBPEL2: Transparent Autonomization in Business Processes through Dynamic Proxies," in *Proceedings of the 8th International Symposium on Autonomous Decentralized Systems (ISADS'07), Sedona, Arizona*, 2007.

[8] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustdar, "Towards Recovering the Broken SOA Triangle – A Software Engineering Perspective," in *Proceedings of the 2nd International Workshop on Service Oriented Software Engineering (IW-SOSWE'07), Dubrovnik, Croatia*, 2007, pp. 22–28. [Online]. Available: http://doi.acm.org/10.1145/1294928.1294934

[9] 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. [Online]. Available: http://doi.acm.org/10.1145/1385989.1386004

[10] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar, "End-to-End Versioning Support for Web Services," in *Proceedings of the International Conference on Services Computing (SCC'08), Honolulu, Hawaii, USA*. IEEE Computer Society, July 2008, pp. 59–66. [Online]. Available: http://dx.doi.org/10.1109/SCC.2008.21

[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, Sept. 2008.

[12] P. Leitner, F. Rosenberg, and S. Dustdar, "DAIOS – Efficient Dynamic Web Service Invocation," *IEEE Internet Computing*, 2009, forthcoming.

[13] E. Oberortner, U. Zdun, and S. Dustdar, "Domain-Specic Languages for Service-Oriented Architectures: An Explorative Study," in *Proceedings of ServiceWave 2008, Madrid, Spain*, P. Mähönen, K. Pohl, , and T. Priol, Eds. Springer, 2008, pp. 159–170.

[14] 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.

[15] E. Visser, "WebDSL: A Case Study in Domain-Specific Language Engineering," TU Defft, The Netherlands, Tech. Rep. TUD-SERG-2008-023, 2008. [Online]. Available: http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2008-023.pdf

[16] D. Groenewegen and E. Visser, "Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns," in *Proceedings of the International Conference on Web Engineering (ICWE'08) Yorktown Heights, New York*. IEEE Computer Society, July 2008, pp. 175–188.

[17] C. Pautasso and G. Alonso, "The JOpera Visual Composition Language," *Journal of Visual Languages and Computing (JVLC)*, vol. 16, pp. 119–152, 2005. [Online]. Available: http://www.jopera.org

[18] 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.

[19] 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.

[20] 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.

[21] A. Lazovik, M. Aiello, and M. Papazoglou, "Planning and monitoring the execution of web service requests," *Journal on Digital Libraries*, vol. 6, no. 3, pp. 235–246, June 2006.

[22] 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.

[23] P. Leitner, A. Michlmayr, and S. Dustdar, "Towards Flexible Interface Mediation for Dynamic Service Invocations," in *Proceedings of the 3rd Workshop on Emerging Web Services Technology (WEWST'08), co-located with the 6th IEEE European Conference on Web Services (ECOWS'08), Dublin, Ireland*, Nov. 2008, pp. 45–59. [Online]. Available: http://www.inf.unisi.ch/faculty/binder/wewst08/wewst08_proceedings.pdf

[24] 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*, Sept. 2006.

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