Planning and Pricing of Service Mashups

Benjamin Blau, Dirk Neumann, Christof Weinhardt
Institute IISM, University of Karlsruhe, Germany
{blau|neumann|weinhardt}@iism.uni-karlsruhe.de

Steffen Lamparter
Institute AIFB, University of Karlsruhe, Germany
sla@aifb.uni-karlsruhe.de

Abstract

Today's development and provision of commercially used Web services has shifted from providing static and predefined functionality to highly configurable services that can be dynamically combined by customers. This new form of composed services called service mashups integrate functionality of multiple sub-services from decentralized providers. Consequently this leads to a high configuration complexity and presents new challenges in the field of planning and pricing of service offerings such as validation of service configurations as well as price determination. Facing these challenges this paper presents an ontology framework for describing technical as well as economic interdependencies between services and shows how planing and pricing algorithms for dynamic Web scenarios can be implemented. As proof of concept we present the implementation of a Service Mashup Planner and show how this tool can be used to construct complex services considering technical as well as economic service aspects.

1 Introduction

Recently there has been a strong shift towards providing information not just on Web sites for the use of human users, but also as Web services for the direct use and manipulation by applications. This trend is manifested by the rise of an API culture where structured information can be obtained using standard protocols such as REST or SOAP. Some of the most prominent examples are service providers such as Amazon, del.icio.us and Flickr. This new form of programmatic access to information gives rise to new kinds of applications that compose information and services from different sources in an innovative fashion and are called *service mashups*. As an example, the Web site ProgrammableWeb[1] currently categorizes about 2500 service mashups which are composed from over 500 services.

Given such a huge amount of services, determining mashups that meet the business requirements of a customer becomes highly complex. Conceptually the engineering process includes three steps: (*i*) discovery of relevant services, (*ii*) determination of compatible composition and (*iii*) the pricing of composition. The complexity of each step is aggravated by providers offering various service customization possibilities together with complex pricing schemes. As an example, consider a financial service which provides portfolio optimization functionality. Such a service can be constructing by combining appropriate computing, storage and content services as illustrated by Figure 1. For each functionality several different service providers are available: computing power is offered by Amazon EC2 or Grid Computing Utility (network.com), storage is offered by Amazon S3 or Google GDrive, and financial data such as stock quotes can be obtained from Xignite or Reuters Online Web services. However, not all possible mashups might be feasible or desirable. For example, data formats provided by one service might not be acceptable for a second service, the quality of service of a mashup might not be acceptable, or the mashup might be too expensive. The latter is often caused by the inflexible pricing schemes of service providers. For example, Sun is offering computation services over network.com for a fixed fee of $1 per CPU/hour. However, this price appears to be too high, as Sun (`http://www.network.com`) is suffering under extremely meager uptake of their computation service. Within their elastic cloud offering (`http://aws.amazon.com/ec2`) Amazon - in constrast - offers computation services at 10 cents per CPU/hour and additionally charge for bandwidth. Apparently, prices are not fixed, but depend on the way how a service is being used. Implementing such dynamic pricing schemes that account for the structure of the service mashup requires more expressive service descriptions as well as more sophisticated composition algorithms.

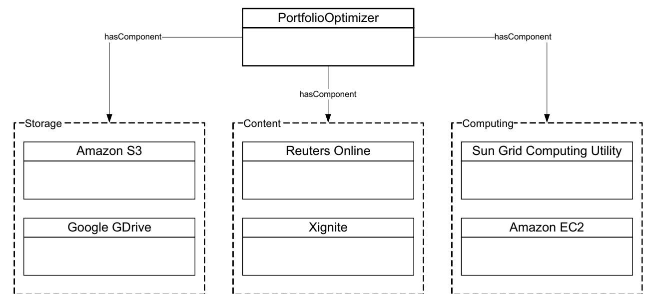

Figure 1. Service Mashup

In order to support the engineering of service mashups this paper proposes an infrastructure for efficient planning and pricing of service mashups. Our contribution is three-

[1] `http://www.programmableweb.com/`

fold. First, we propose an ontology framework which allows to model services as well as their dependencies. A major contribution to the state of the art is the extensive support for economic concepts such as bundling of services. Secondly, we develop an extended form of a path auction for price determination of service mashups as a more efficient mechanism compared to flat fees and pay-per-use pricing schemes. Thirdly, we show how the proposed pricing mechanism can be implemented in a Web environment based on the ontology framework.

The paper is structured as follows. In Section 2 we derive basic requirements for a system that supports planning and pricing of We service mashups and discuss related approaches. In Section 3 we propose an ontology framework which provides primitives for modeling non-functional and economic service descriptions. Based on that framework Section 4 introduces the application of a special form of path auction to determine the price for a service mashup based on offers from decentralized sub-service providers. The goal of the mechanism is to determine a low-cost combination of sub-services and therefore to incentivize providers to reveal their costs for providing requested functionality truthfully. As proof of concept, Section 5 briefly describes our prototype and its components. Section 6 closes with a conclusion and points out further challenges and future work.

2 Requirements and Related Work

In this section, we derive requirements for a system that supports engineering complex mashups and review to what extend existing work already meets the identified requirements.

While web tools such as Yahoo! Pipes[2] provide means for integrating certain information services, similar tools for a broader range of services considering aspects like QoS and economic properties are still missing. To support such aspects more expressive descriptions of service offerings and requrests are required.

Requirement 1 (Expressive Descriptions) *Provide an expressive description of service offers and requests, which can be used to semantically describe information about the capability/functionality of a service, which at least includes a description of the information involved in a service invocation (i.e. input and output information) and the different possible configurations of a service with the corresponding price (e.g. different service levels with different prices).*

There is already a substantial amount of work in the area of service discovery considering different aspects of a service. Classically a service is considered relevant if it takes the right input types and returns the right output types. Depending on the expressivity of the language in which these types are defined one can distinguish between rather syntactic approaches such as UDDI on the hand and semantic approaches such as OWL-S, WSMO and SAWSDL on the other hand. The latter use ontologies to formally describe information required and returned by a service and improve several shortcomings of purely syntactic solutions such as data integration from heterogenous sources or pure recall and precision.

Since input and output is often not sufficient to completely define the functionality of a service, several extensions have been proposed: As an example, OWL-S, WSMO and several other approaches such as [10] allow for semantic descriptions of pre- and postconditions/effects, while [6, 1] captures the behavior of a service by more detailed description of the service's internal process. While all the approaches mentioned above do not directly consider non-functional properties of services in the discovery processes, other approaches like [7, 16] add expressiveness with respect to quality of service and allow the specification of service levels together with corresponding prices. Thereby, they enable the specification of configurable services. However, these approaches focus only on single services and thus cannot capture all kinds of dependencies between services (e.g. expressing service bundles or the incompatibility of two services cannot be done explicitly) which is essential information to enable composition of services.

Requirement 2 (Composition Support) *Support of the service planning process by analyzing service descriptions and proposing suitable service combinations and configurations to the customer. In this context, service mashups that lead to incompatibilities, do not provide the right functionality, or miss certain quality targets should be automatically filtered out, which requires a description of service dependencies in terms of compatibility as well as pricing, and contextual information on service interrelations.*

In the composition step services are composed to a single mashup, which might include passing the output of one service through to the input of another service, integrating data from different sources, translating data formats and schemas, etc. As finding mashups that realize a certain functionality becomes increasingly cumbersome as the number of services grows, machine support for finding valid and suitable compositions is required. In order to tackle this complexity a wide range of different methods have been proposed for automatically calculating the compositions. Most of these approaches rely on AI planning algorithms and apply backward chaining to derive suitable compositions from a certain goal. Such an approach is, for example, presented in [17] for stateful and in [22] for stateless services. While these approaches consider semantic descriptions of input and output, they do not consider dependency between services. This problem is addressed by composition on process level, where the (temporal) behavior of a service is also considered. Such approaches are presented, e.g., in [6]. However, these approaches are not sufficient since they only consider desired functionality as composition goal and disregard quality of service. Approaches for QoS-based composition are presented in [9, 24, 15]. Since they require predefined aggregation rules for all QoS-attributes, they are only partly suitable for full automation in distributed environments. In addition, they have very limited support for service dependencies (such as bundles) and provide only fixed pricing mechanisms.

[2] http://pipes.yahoo.com

Requirement 3 (Pricing Mechanism) *The pricing mechanism has to support a highly flexible, decentralized environment, where the different providers are self interested and own private information. Therefore, the pricing mechanism has to be incentive compatible in order to make sure that all services reveal their true valuation and should also allow multiple configurations per service. Since in a competitive environment potential sub-service providers change rapidly, the pricing mechanism must provide easy execution, iterative price determination and low computational complexity. In order to give consideration to the variety of service properties the mechanism should be capable of multiattribute offers and corresponding requesters' preferences.*

After determining suitable service mashups, one has to decide which of them is most appropriate. Therefore, prices for a set of possibly customizable services from different sources have to be determined. This could be already quite challenging. Some service providers like Amazon offer tools for calculating prices given a certain configuration. For finding the best service with the best configuration service selection algorithms have been proposed in [8, 16, 15] that rely on explicit preference information of the customer. The problem with these approaches is on the one hand that they are restricted to the selection of single services and on the the other hand that they support only fixed pricing mechanisms featuring simple flat fees and pay-per-usage price models. Focusing on more complex graph-like interactions, a lot of research has been done recently in the field of path auctions [4, 14, 13, 21]. However, these forms of path auctions have mostly been applied to problems in supply chain management and network routing.

Recapitulating, existing methods and tools for engineering service mashups lack expressive descriptions of economic service properties as well as mechanisms that allow an efficient planning and pricing of a complex mashup. The following section introduces our ontology framework addressing these issues.

3 Planning of Service Mashups

The process of commercially planning a service mashup that facilitates a bundle of sub-services is crucial and needs a lot of expert knowledge from a technical and business perspective. A service provider faces issues like incompatibility, domain specific configuration constraints and inconsistency. Therefore it is necessary to support the process of service mashup planning by incorporating expert knowledge and providing basic design patterns for modeling service infrastructures.

In this section we propose a three layered ontology framework which meets the requirement for expressive service description as stated in the last section. Before introducing the framework in Section 3.2, Section 3.1 presents the fundamentals of the ontology formalism used.

3.1 Ontology Formalism

As a formalism to represent our ontology framework we use OWL. OWL is an ontology language standardized by

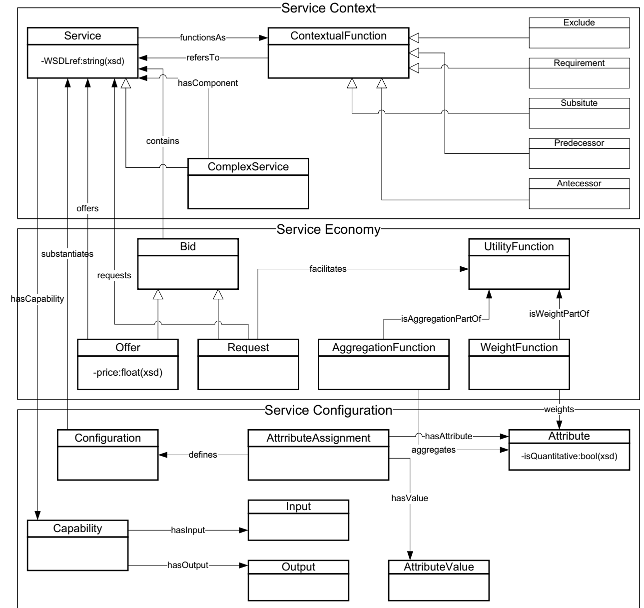

Figure 2. Generic Service Ontology

the World Wide Web Consortium (W3C) [23] and is based on the description logic (DL) formalism [5]. Due to its close connection to DL it facilitates logical inferencing and allows to derive conclusions from an ontology that have not been stated explicitly. We briefly review some of the modeling constructs of OWL using its DL-syntax. The main elements of OWL are *individuals*, *properties* that relate individuals to each other and *classes* that group together individuals which share some common characteristics. Classes as well as properties can be put into subsumption hierarchies. Furthermore, OWL allows for describing classes in terms of complex *class constructors* that pose restrictions on the properties of a class. For example, the statement $\mathsf{BigCity} \sqsubseteq \exists\ \mathsf{isConnectedTo.Highway}$ describes the class of big cities, which are connected to some Highway. Subclass relationship can be expressed by a statement like $\mathsf{BigCity} \sqsubseteq \mathsf{InterestingCity}$, saying that any big city is also interesting.

For the reader's convenience we illustrate our ontology in UML notation where UML classes correspond to OWL concepts, UML associations to object properties, UML inheritance to sub-concept relations, UML dependencies to OWL class instantiations and UML attributes to OWL datatype properties.

3.2 Ontology Framework

We propose an ontology framework for describing Web services which is structured in three parts: The *Generic Service Ontology*, the *Domain Specific Service Ontology*, and the *Service Instance Layer*.

The first part represents the ***Generic Service Ontology*** which defines relevant concepts to specify services in general (independent from a concrete application scenario)(see Figure 2). The Generic Service Ontology functions as a

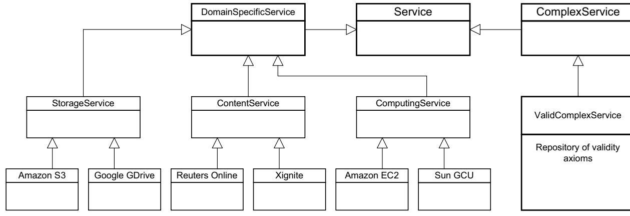

Figure 3. Domain Specific Service Ontology

ontology design pattern for modeling service mashups in a standardized manner. It is structured in three parts: *Service Context*, *Service Economy* and *Service Configuration*. Concepts in the service context part support the description of contextual information on service interrelations. E.g. a service can function as a predecessor referring to another service. At the same time it can be an enhancement for this particular service. To assure the correctness of service interrelations, the contextual information is supported by a set of rules which is explained in the Section 3.3. The second part of the ontology combines concepts related to economics which play a central role for service mashup price determination. *Offers* and *Requests* are modeled as a specific type of *Bid*. An offer offers a concrete service instance at a certain price. A request specifies a service as well as a utility cap which is at least expected from the service execution. The concept *AggregationFunction* defines an operation to aggregate a certain type of attribute (e.g. an operation for a quantitative attribute could be a simple addition). Concepts for aggregating service properties have been discussed in [9, 2]. The concept *WeightFunction* represents weights for different types of attributes which implicitly represents the requestors preferences. Thirdly, the service configuration part represents concepts for describing service capabilities and instance configurations. A *Configuration* is defined by an assignment of *Attributes* and their *AttributeValues* which form concrete service instances. The attribute concept specifies the semantic type of service property (e.g. response time, encryption mode) which can either be quantitative or qualitative. The *Capability* concept is divided in an *Input* and *Output* part.

In order to apply the framework in a concrete setting, a ***Domain Specific Service Ontology*** is required that represents domain specific knowledge on the service mashup and its sub-services for a given scenario as depicted in Figure 3. As domain services are represented as sub-classes of the central *Service* concept in the Generic Service Ontology they inherit predefined relations. Besides domain services, expert knowledge about service compatibility and interrelation is captured by a concept called *ValidComplexService*[3] which is a subclass of the *ComplexService* concept in the Generic Service Ontology.

The third part of the ontology is the ***Service Instance Layer***. It represents the result of the planning process as an instance of the *ComplexService* concept. This instance describes a concrete service mashup, its possible sub-service configurations and their interrelation.

[3]We use the term complex service as an equivalent to service mashup.

3.3 Planning Support

The Generic Service Ontology introduces the concept *ContextualFunction* and more concrete sub-concepts like *Exclude*, *Requirement*, *Substitute*, *Predecessor* and *Antecessor*. These concepts and their impact on the service planning process are specified by rules. These rules assure that once a contextual relation between services is defined, mashups of different services can be verified using logical inferencing.

For the declarative formulation of matching directives in form of rules, we require additional modeling primitives not provided by OWL. We use the Semantic Web Rule Language (SWRL) [12] which allows us to combine rule approaches with OWL. We restrict ourselves to a fragment of SWRL called DL-*safe* rules [19], which is more relevant for practical applications due to its tractability and support by OWL inference engines. For the notation of rules in this paper we rely on a standard first-order implication syntax.

For example, a service B functions as a requirement referring to a service A and service A is part of a complex service C. This implies that service B must also be a component of complex service C to assure a flawless execution. These contextual rules also work the other way around. If, for example, capabilities of a service A are subsumed by capabilities of a service B, that implies that service B functions as a substitute for service A.

The following list of rules is a sub-set of the rule repository in the Generic Service Ontology, which define the semantics of *Exclude*, *Requirement*, *Substitute*, and *Pre-/Antecessor*.

Exclude Service(?x) ∧ Service(?y) ∧ ComplexService(?z) ∧ hasComponent(?z, ?x) ∧ Exclude(?e) ∧ functionsAs(?x, ?e) ∧ refersTo(?e, ?y) → ¬ hasComponent(?z, ?y)

Service(?x) ∧ Service(?y) ∧ Exclude(?e) ∧ functionsAs(?x, ?e) ∧ refersTo(?e, ?y) → functionsAs(?y, ?e) ∧ refersTo(?e, ?x)

Requirement Service(?x) ∧ Service(?y) ∧ ComplexService(?z) ∧ hasComponent(?z, ?x) ∧ Requirement(?r) ∧ functionsAs(?x, ?r) ∧ refersTo(?r, ?y) → hasComponent(?z, ?y)

Substitute Service(?x) ∧ Service(?y) ∧ hasCapability(?x, ?a) ∧ hasCapability(?y, ?b) ∧ subsumes(?a, ?b)[4] → Substitute(?s) ∧ functionsAs(?x, ?s) ∧ refersTo(?s, ?y)

Service(?x) ∧ Service(?y) ∧ Substitute(?s) ∧ functionsAs(?x, ?s) ∧ refersTo(?s, ?y) → functionsAs(?y, ?s) ∧ refersTo(?s, ?x)

Pre-/Antecessor Service(?x) ∧ Service(?y) ∧ Predecessor(?p) ∧ functionsAs(?x, ?p) ∧ refersTo(?p, ?y) → Antecessor(?a) ∧ functionsAs(?y, ?a) ∧ refersTo(?a, ?y)

The central supporting component for model validation is integrated in the Domain Specific Ontology. The concept *ValidComplexService* is specified by a set of DL-axioms that form the set of valid service configuration bundles. A modeled service mashup and its sub-services are added as an individual of the *ComplexService* concept. The process of model validation is twofold.

First, the ontology including the created individuals is checked for consistency which is a standard reasoning task.

[4]subsumes is a SWRL buit-in that verifies if a concept A subsumes a concept B meaning that the interpretation of concept B is a subset of the interpretation of concept A

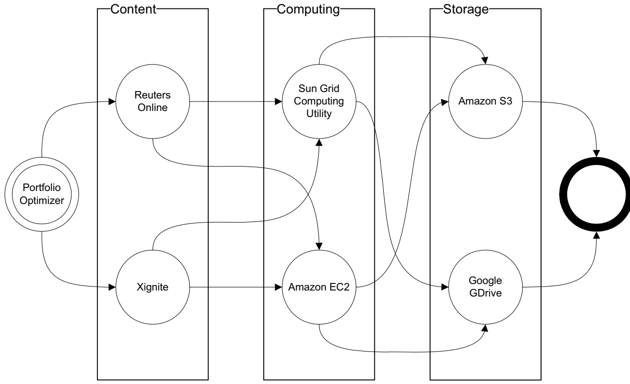

Figure 4. Service Mashup Graph

As a second step a another task is executed that infers if the modeled service individual is also an instance of the *ValidComplexService* concept. If both validation steps yield positive results the planned complex service is considered to be valid.

The following example shows an axiom defining the concept *ValidComplexService*. If an instance of the concept *AmazonEC2* functions as a predecessor referring to an instance of the concept *AmazonS3* its configuration must have an attribute *AmazonUserID*. Instances that satisfy this axiom are inferred to be an instance of the concept *ValidComplexService*.

ValidComplexService ⊑ ComplexService
ValidComplexService ⊑
(∀ hasComponent (AmazonEC2 ⊓ (∃ functionsAs (Predecessor ⊓ (∃ refersTo AmazonS3))) ⊓
(∃ isSubstantiatedBy (Configuration ⊓ (∃ isDefinedBy (AttributeAssignment ⊓ (∃ hasAttribute AmazonUserID ⊓ ∃ hasValue{UserID})))))))

Once the Domain Specific Ontology including the rule sets and validation axioms are defined, the framework functions as a design pattern for further change management and reconfiguration. The final result of a service mashup planning process is a graph which defines all suitable subservice configurations and their reasonable interrelations. Recalling our scenario, Figure 4 shows a simplified output of the planning process of our portfolio optimization mashup service.

In summary, we proposed an ontology framework that provides capabilities for expressive service descriptions including service input and output capabilities, configurations, economic aspects as well as dependencies and contextual information on service interrelations which satisfies Requirement 1. Furthermore we presented rule-based modeling support for contextual service functionality and a validation concept that assures consistency of designed service mashups which satisfies Requirement 2.

4 Pricing of Service Mashups

In this section we demonstrate the application of a path auction to solve the price determination problem for service mashups based on the results of the planning process. We implement a Vickrey-Clarke-Groves (VCG) mechanism to incentivize self-interested sub-service providers to reveal their costs truthfully.

In analogy to procurement scenarios, we suggest to establish a reverse auction for getting the sub-services of the mashup in order to drive the prices for the sub-services down to a reasonable level. As long as service providers can cover their costs, including interest on their invested capital, service providers are expected to participate in this kind of reverse auction. Challenges in this scenario are twofold: Firstly, the mashup can be used by different combinations of services, where the number and type of services of each combination can be different. Secondly, not all services can be orchestrated into a mashup. This makes the use of a series of mutliple independent single reverse auctions inapplicable due to the service dependencies.

Thus, we propose to apply a path auction for the mashup pricing problem. The path auction has been initially introduced by Nisan and Ronen in their seminal paper on algorithmic mechanism design [20]. The path auction has been proposed for the use in multi-domain routing [11], but is highly debated [18]. The application of path auctions in mashup pricing seems far more promising, as we will argue below.

4.1 The Path Auction

From the service planning, we obtained a network of sub-services, which laid out the building plan of the mashup. As in Figure 4, the mashup structure can be represented as a directed disjoint graph $G = (V, E)$ with an initial node v_s and a final node v_f. Let F denote the feasible set of all paths from source to sink. Obviously, F is a result of the planning phase and can be subsequently used for establishing a reverse path auction. A node $v_i \in V$ represents a sub-service that is owned by an autonomous service provider $s \in S$. Each edge $e_{ij} \in E$ in the graph denotes the invocation of sub-service j by i. The costs c_{ij} incurred by the use of a sub-service depend on the antecedent sub-service that invokes this sub-service. These costs are attached to each edge and are only known to service provider k.

To install a path auction each service provider places an offer $b_{ij} \in B$ for each edge she owns. The path auction is intended to find out the lowest cost path, i.e. that configuration of the mashup that causes the lowest costs. Sub-service providers which are on the winning path receive a payment according to a payment function $P : B \to \mathbb{R}$. Let $I^k(c_{ij})$ be the indicator function for the winning path from i to j. $I^k(c_{ij}) = 1$, if sub-service k is on the winning path and 0 if not. Moreover we assume that

- (G, F) is common knowledge to mashup requester and sub-service providers.
- There is no monopoly situation such that the same edge is in every feasible set F.

- Sub-service providers act rationally meaning they try to maximize their utility.
- Sub-service providers have quasi-linear utilities $U_k = \sum_{i,j\in V}(p^k_{ij} - I^k(c_{ij})c_{ij})$.

The auction is conducted as follows:

1. The mashup requester announces his aggregated utility cap $\bar{U}$, which denotes the maximum price it will pay for the entire mashup.
2. Any sub-service provider k places bids o_{ij} for each edge e_{ij} that is ending in k.
3. The mashup requester calculates his aggregated utility $U_{Mashup} = \bar{U} - \sum_{i,j\in V}(p^k_{ij})$ for any path and computes the winning path, which corresponds to the Low Cost Path (LCP) from source to sink using a well-established shortest path algorithm like Dijkstra.
4. The winning path is chosen by the mashup requester if $U_{Mashup} \geq 0$, otherwise the mashup requester does not purchase the mashup.
5. If the calculated winning path is chosen, the sub-service providers which own an edge that is part of the winning path receive their payments

The winning path equals the LCP if the sub-service providers reveal their costs truthfully, i.e. $b_{ij} = c_{ij} \forall i, j \in V$. Truthful revelation of costs can be induced by the Vickrey-Clarke-Groves (VCG) scheme, where service provider k gets a premium that reflects the utility gain by service provider k being present [11]. That is, service provider k receives payment for providing service j revoked from i by $c_{ij}I^k(c_{ij}) + \sum_{r\in V\setminus k} I^r(c_{ij})c_{ij} - \sum_{r\in V} I^r(c_{ij})c_{ij}$ which amounts to the costs of performing the service plus the difference between the LCP without service provider k present and the LCP. The VCG yields the valuable property of inducing truth-telling as weakly dominant strategy and leaves all participants with a non-negative utility. The budget is not balanced, but this is uncritical, as the payments are directed from the mashup requester to the service providers.

4.2 Example

Suppose now the services have the following (arbitrary) prices. Accessing Reuters data costs \$12. The following computation job can be performed by using Sun's network.com 8 CPU/hours for \$8 or for \$9 by Amazon. As a storage server Amazon's S3 can be used which costs \$2 if revoked by network.com or \$3 by Amazon. The set of bids is depicted in Figure 5.

Clearly, the winning path of this mashup is the use of Xignite, Sun and Amazon S3 amounting to a total of \$19. The price for this mashup is generated by the sum of all individual VCG prices. The Xignite service costs $\$14$ plus the premium. Without Xignite being present the winning path would be Reuters, Sun and S3 which cost \$22. Thus the price for Xignite is $\$14 + \$22 - \$19 = \$17$. Likewise, the VCG price for Sun is $\$9 = \$8 + \$20 - \$19$ and $\$4 = \$2 + \$21 - \$19$ for Amazon S3. Thus, the mashup price amounts to \$30.

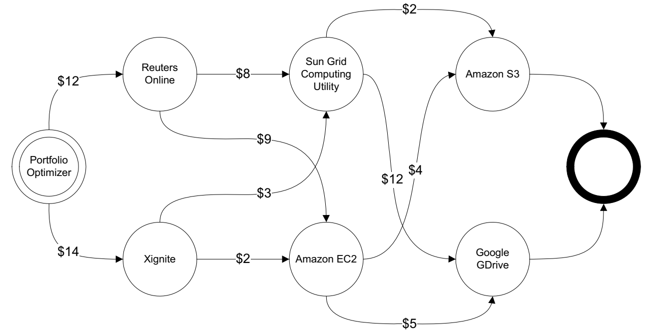

Figure 5. Bids in a Service Mashup Graph

The VCG mechanism is a good starting point for path auctions and perfectly fits into the planning scenario. However, one has to assume sufficient alternative services in the system in order to avoid unacceptably high prices. In those cases the mashup requester will not purchase the sub-services [4]. For the future we plan to extending the mechanism in order to handle also scenarios with low fungibility.

4.3 Multiattribute Extension

Choosing a set of sub-services only based on prices they charge might not be sufficient to satisfy the requesters' requirements with regard to non-functional aspects and quality of service. Therefore, we introduce an multiattribute extension to the proposed price determination mechanism which allows requesters to specify there quality preferences. This *Extended Path Auction* is executed as follows:

1. The mashup requester announces her aggregated utility cap $\bar{U}$ in addition to a price cap $\bar{P}$.
2. Any sub-service provider k places bids b_{ij} for each edge e_{ij} that is ending in k. Each bid is a tuple $(a^1_{ij}, \ldots, a^n_{ij}, p_{ij})$ with $a^1_{ij}, \ldots, a^n_{ij}$ describing a set of attribute values and price for the service instance offered.
3. The mashup requester calculates her aggregated price $P_{Mashup} = \sum_{i,j\in V}(p^k_{ij})$ and aggregated utility U_{Mashup} for any path and computes the winning path, which corresponds to the Low Cost Path (LCP) from source to sink using a well-established shortest path algorithm like Dijkstra.
4. The winning path is chosen by the mashup requester if $U_{Mashup} \geq \bar{U} \wedge P_{Mashup} \leq \bar{P}$, otherwise the mashup requester does not purchase the mashup.
5. If the calculated winning path is chosen, the sub-service providers which own an edge that is part of the winning path receive their payments.

4.4 Processing of Orders

The representation of requester's preferences and the computation of aggregated utilities is not a trivial task. The following section proposes an algorithm-based solution for combining technical and economical service descriptions from Section 3.2 with proposed mechanism for implementation as a web-based application.

The formal notation from the previous section can be directly mapped to ontology constructs. For example, the set of offers B is represented by the concept *Offer* in the ontology, i.e. an instance of *Offer* corresponds to an element $b \in B$. We are therefore able to implement the pricing mechanism directly as operations on the knowledge base.

Once an offer arrives it is stored in an offer repository. Then the mashup requester computes the overall utility of the set of offers using the Algorithm 1. For a given domain ontology (i.e. a predefined set of attributes A_a)) the algorithm can also be represented directly as a query (e.g. using SPARQL).

Algorithm 1 ComputeMashupUtility

Require: B
1: **for all** attribute concepts A_a in each $b \in B$ **do**
2: $\quad a = \mathsf{hasValue}.(\mathsf{hasAttribute}^{-1}.A_a)$
3: $\quad U = add(U, ComputeUtility(a, B))$
4: **return** U

To determine the overall utility *ComputeMashupUtility* invokes the function *ComputeUtility* for each service attribute. The *ComputeUtility* function retrieves the weight of an attribute from the ontology and aggregates the utility of attributes values along the service composition. Therefore, the *AggregateProperty* function determines the individual aggregation function for each type of attribute in the ontology and aggregates quantitative and qualitative service properties over all offers as shown in Algorithm 2. For example, service properties can be "response time" and "encryption mode". The concept *AggregationFunction* defines an individual operation for each type of attribute. The attribute response time is a quantitative property and is aggregated additively. Encryption mode on the other hand is a qualitative property. Therefore it makes sense to define an aggregation operation like the logical AND because if only one sub-service does not provide encryption capabilities the service mashup is not secured which yields a utility of 0 from a service mashup provider perspective.

Algorithm 2 AggregateProperty

Require: a, B
1: $A_a = \mathsf{hasAttribute}.(\mathsf{hasValue}^{-1}.a)$
2: $\oplus = \mathsf{aggregates}^{-1}.A_a$
3: **for all** $b \in B$ that entail attribute A_a **do**
4: $\quad agg = agg \oplus a \in b$
5: **if** $\mathsf{isQuantitative}.A_a$ **then**
6: $\quad$ **return** $div(a, agg)$
7: **else**
8: $\quad$ **return** agg

The calculation of the lowest cost path is performed by Algorithm 3. The algorithm is a modification of the well-known Dijkstra algorithm for computing the shortest path in a graph. The runtime complexity is $O(|V|^2 + |E|)$ with V denotes the set of vertexes and $|E|$ the set of edges. If the graph satisfies certain conditions the searching step can be implemented more efficiently decreasing the complexity to a logarithmic level [3].

Algorithm 3 DetermineShortestPath

Require: G, v_s, v_f, B
1: **for all** $v \in G$ **do**
2: $\quad price[v] = \infty$
3: $\quad previous[v] = \emptyset$
4: $\quad push(v, Q)$
5: $price[v_s] = 0$
6: **while** $Q \neq \emptyset$ **do**
7: $\quad u = retreiveMin(Q)$
8: $\quad$ **if** $u = v_f$ **then**
9: $\quad\quad$ **while** $previous[u] \neq \emptyset$ **do**
10: $\quad\quad\quad addAtFirstPosition(u, W)$
11: $\quad\quad\quad u = previous[u]$
12: $\quad\quad$ **return** W
13: $\quad$ **for all** neighbors v of u **do**
14: $\quad\quad$ **if** $price[u] + p_{uv} < price[v]$ **then**
15: $\quad\quad\quad price[v] = price[u] + \mathsf{hasPrice}^{-1}.o_{uv}$
16: $\quad\quad\quad previous[v] = u$

In summary, we proposed an application of a multiattribute path auction with an incentive compatible mechanism that supports service mashup providers in determining a low-cost combination of sub-services from a set of decentralized self-interested sub-service providers in a flexible environment with asymmetric information. By incorporating semantic service descriptions into the auction, it can be applied in web-based applications. In summary this satisfies Requirement 3.

5 Prototype

Proposed ontology framework and concepts to support modeling of service mashups as described in Section 3 have been implemented as an Eclipse RCP Application facilitating the Eclipse Modeling Framework (EMF) and Eclipse Graphical Modeling Framework (GMF). The integration of ontologies and reasoning functionality is realized through Jena[5] and the Pellet OWL Reasoner[6].

The service mashup planner provides graphical planning support for modeling service interrelations based on a given meta-model. The designer can draw the sub-service structure and interrelations using a set of graphical tools.

Anytime during the planning process the developed model instance can be validated against the underlaying ontology framework. The validation step provides detailed information on pattern violations or inconsistencies. The result of a planning process is a sub-service graph that specifies reasonable sub-service configurations and their interrelations as part of an overall service mashup.

The prototype is part of an industry project in cooperation with a major international IT service provider and hardware manufactorer. The prototype has been evaluated and will form the blueprint for the improvement of a commercial resource planning tool. A online demonstration is available at `http://www.iw.uni-karlsruhe.de/smp`.

[5] `http://jena.sourceforge.net/`
[6] `http://pellet.owldl.com/`

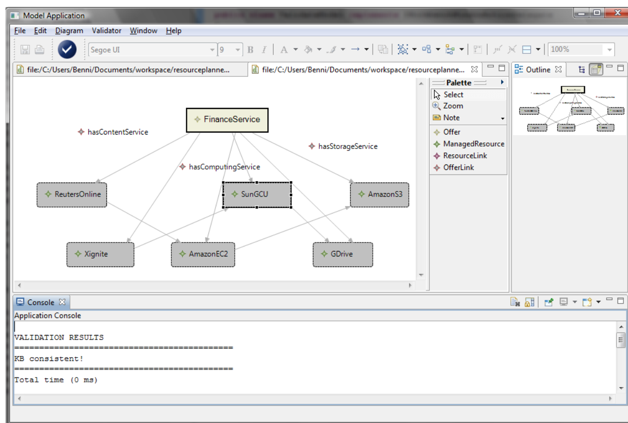

Figure 6. Service Mashup Planner GUI

6 Conclusion

The main motivation of this work was missing support for non-functional aspects (particularly economic aspects) as well as flexible pricing mechanisms in the existing concepts and tools. We therefore proposed an expressive ontology framework featuring concepts for describing non-functional and economic related service properties as well as service capabilities, configurations and contextual information on interrelations. Thus, the ontology can be used for externalizing expert knowledge and automated validation of mashups. To support flexible pricing of mashups we introduced an extended path auction for price determination of service mashups. The mechanism supports the service mashup provider in finding a low-cost combination of sub-service fulfilling an overall mashup functionality. Therefore the mechanism incentivizes self-interested and decentralized sub-service providers to reveal their costs truthfully. By combining the auction mechanisms with semantic service descriptions, we provided algorithms for implementing such auctions in the Web. As proof of concept a prototypical mashup planing and pricing tool has been implemented based on proposed ontology framework and pricing mechanism.

References

[1] S. Agarwal. *Formal Description of Web Services for Expressive Matchmaking*. PhD thesis, Fakultät für Wirtschaftswissenschaften, Universität Karlsruhe (TH), May 2007.

[2] S. Agarwal and S. Lamparter. User Preference Based Automated Selection of Web Service Compositions. *ICSOC Workshop on Dynamic Web Processes*, pages 1–12, 12 2005.

[3] R. Ahuja, K. Mehlhorn, J. Orlin, and R. Tarjan. Faster Algorithms for the Shortest Path Problem. *Journal of the Association for Computing Machinery*, 37(2):213–223, 1990.

[4] A. Archer and Éva Tardos. Frugal Path Mechanisms. *ACM Trans. Algorithms*, 3(1):3, 2007.

[5] F. Baader, D. Calvanese, and D. L. McGuinness. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2007.

[6] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic Services Composition based on Behavioral Descriptions. *Int. J. of Cooperative Information Systems (IJCIS)*, 14(4):333376, 2005.

[7] M. Bichler and J. Kalagnanam. Configurable Offers and Winner Determination in Multi-attribute Auctions. *European Journal of Operational Research*, 160(2):380–394, 2005.

[8] P. A. Bonatti and P. Festa. On Optimal Service Selection. In *Proc. of the 14th Int. Conf. on WWW*, pages 530–538, New York, NY, USA, 2005.

[9] J. Cardoso, A. Sheth, J. A. Miller, J. Arnold, and K. Kochut. Quality of Service for Workflows and Web Service Processes. *Journal of Web Semantics*, 1(3):281–308, 2004.

[10] I. Constantinescu, W. Binder, and B. Faltings. Flexible and Efficient Matchmaking and Ranking in Service Directories. In *IEEE Int. Conf. on Web Services (ISWC'05)*, pages 5–12, Orlando, USA, 2005.

[11] J. Feigenbaum, C. Papadimitriou, R. Sami, and S. Shenker. A BGP-based Mechanism for Lowest-cost Routing. *Distributed Computing*, 18(1):61–72, 2005.

[12] I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. *W3C Submission*, 21, 2004.

[13] N. Immorlica, D. Karger, E. Nikolova, and R. Sami. First-price Path Auctions. *Proceedings of the 6th ACM conference on Electronic commerce*, pages 203–212, 2005.

[14] A. Karlin, D. Kempe, and T. Tamir. Beyond VCG: Frugality of Truthful Mechanisms. *Proc. of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 615–626, 2005.

[15] U. Küster, B. König-Ries, M. Klein, and M. Stern. DIANE - A Matchmaking-Centered Framework for Automated Service Discovery, Composition, Binding and Invocation. In *Proc. of 16th Int. WWW 2007*, Banff, Canada, 2007.

[16] S. Lamparter, A. Ankolekar, S. Grimm, and R. Studer. Preference-based Selection of Highly Configurable Web Services. In *Proc. of the 16th Int. World Wide Web Conference (WWW'07)*, pages 1013–1022, Banff, Canada, May 2007.

[17] F. Lécué and A. Léger. A Formal Model for Semantic Web Service Composition. In *ISWC the 5th International Semantic Web Conference*, pages 385–398, November 2005.

[18] P. Maille and B. Tuffin. Why VCG Auctions Can Hardly be Applied to the Pricing of Inter-domain and Ad Hoc Networks. *Proc. of the 3rd EURO-NGI Conf. on Next Generation Internet Networks*, 2007.

[19] B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with Rules. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(1):41–60, 2005.

[20] N. Nisan and A. Ronen. Algorithmic Mechanism Design. *Games and Economic Behavior*, 35(1-2):166–196, 2001.

[21] A. Ronen and R. Talisman. Towards Generic Low Payment Mechanisms for Decentralized Task Allocation. *Proc. of the 7th Int. IEEE Conf. on E-Commerce Technology*, 2005.

[22] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. S. Nau. HTN planning for web service composition using SHOP2. *Journal of Web Semantics*, 1(4):377396, 2004.

[23] W3C. Web Ontology Language (OWL). `http://www.w3.org/2004/OWL`, 2004.

[24] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.