Towards Custom Cloud Services

Using Semantic Technology to Optimize Resource Configuration

Steffen Haak and Stephan Grimm

Research Center for Information Technology (FZI)
Haid-und-Neu-Str. 10-14
D-76131 Karlsruhe, Germany
{haak,grimm}@fzi.de

Abstract. In today's highly dynamic economy, businesses have to adapt quickly to market changes, be it customer, competition- or regulation-driven. Cloud computing promises to be a solution to the ever changing computing demand of businesses. Current SaaS, PaaS and IaaS services are often found to be too inflexible to meet the diverse customer requirements regarding service composition and Quality-of-Service. We therefore propose an ontology-based optimization framework allowing Cloud providers to find the best suiting resource composition based on an abstract request for a custom service. Our contribution is three-fold. First, we describe an OWL/SWRL based ontology framework for describing resources (hard- and software) along with their dependencies, interoperability constraints and meta information. Second, we provide an algorithm that makes use of some reasoning queries to derive a graph over all feasible resource compositions based on the abstract request. Third, we show how the graph can be transformed into an integer program, allowing to find the optimal solution from a profit maximizing perspective.

1 Introduction

In the last decades, most companies regarded their IT systems as a necessary but unimportant part of their business models. Investments into IT infrastructure were cost- rather than quality-driven and long product life cycles made investments long-term rather than flexible. However the Internet's increasing importance for global business, emerging new paradigms like Service Oriented Architectures (SOA) and Cloud computing, and the ever increasing competition fostered through globalization has made many companies rethink their IT strategy. Even in traditional industries, IT no longer just acts as a supporting unit - in many cases IT has become a strategic competitive factor. The challenge has become even harder, as product and service life cycles become shorter and adaptation to market changes needs to be more agile as ever before [6]. Naturally this challenge is encountered by all parts of modern business, putting high demands on flexibility and agility on a company's supporting IT services.

G. Antoniou et al. (Eds.): ESWC 2011, Part II, LNCS 6644, pp. 345–359, 2011.

Cloud computing (and its SaaS, PaaS and IaaS offerings) promises to be a solution for cutting costs in IT spending while simultaneously increasing flexibility. According to thecloudmarket.com [1], there already exist over 11.000 different preconfigured Amazon EC2 images from various providers, that can be run as virtual appliance on Amazon's Elastic Compute Cloud (EC2). The large variety indicates the importance of custom service offers. However, the customer has little support in the technical and economic decision process for selecting an appropriate image and deploying it on the Cloud. It is neither guaranteed, that there finds an image that is configured exactly according to the customer needs.

Apparently there is a great business opportunity for Cloud providers who manage to offer Custom Cloud Services tailored to their customers' needs. Transferring the selection and deployment process to the provider simplifies the customer's decision process significantly. In such a scenario, the provider faces the challenging task of automatically finding the optimal service composition based on the customer request, from both a technical and an economic view. A customer requesting a database service having for example only preferences on the underlying operating system or the available storage space leaves a lot of room for economic optimization of the service composition. A MySQL database running on a Linux-based virtual machine (VM) might be advantageous over the more costly Oracle alternative on a Windows-based VM.

Custom Cloud Services, as referred to in this paper, are compositions of commercial off-the-shelf software, operating system and virtualized hardware, bundled to offer a particular functionality as requested by a customer. The functional requirements usually leave many choices when choosing required resources from groups that offer equal or similar functionalities as in the above mentioned example. Finding the optimal choice involves many different aspects, including (among others) technical dependencies and interoperability constraints, customer preferences on resources and Quality-of-Service (QoS) attributes, capacity constraints and license costs. In a sense, it is a configuration problem as known from typical configurators for products like cars, etc. However, stated as integer programming or constraint optimization problem [15,7], it is underspecified as not all configuration variables (the set of required service resource types) can be clearly specified ex ante from the customer request. The missing variables however can be derived dynamically as they are implicitly contained in the resources' dependencies. For example, a customer request for a CRM software might implicitly require an additional database, some underlying operating system on some virtualized hardware.

As mentioned before, traditional linear or constraint programming techniques require the knowledge of all variables. In order to overcome this problem, we propose an ontology-based approach for a convenient and standardized knowledge representation of all known Cloud service resources, allowing to derive the complete configuration problem by subsequently resolving resource dependencies.

The remainder of this paper is structured as follows. Section 2 describes an example use case, which helps us to derive requirements for the designated

framework. These requirements are also used for a qualitative evaluation of our approach and to distinguish it from related work.

In Section 3 we describe our main contribution. We start by describing our understanding of functional requirements, which serve as input for the ontology-based optimization framework. We then propose the usage of a three-fold ontology system to serve as knowledge base. We distinguish between a generic service ontology, that contains the meta concepts provided in this paper, a domain ontology that makes use of these concepts and contains the actual knowledge about known infrastructure resources, and a result ontology that is used to represent a graph, spanning a network over different choices based on abstract dependency relations that can exist between different resources (e.g. that every application needs some operating system). We formally describe this dependency graph and show how it can be derived algorithmically, making use of different queries to the ontology system. Further we show how this graph can be used to obtain all feasible infrastructure compositions. In addition, we show how the graph can be transformed into an integer program for finding the profit optimal configuration with respect to customer preferences and costs. For knowledge representation, we make use of the Semantic Web ontology and rule languages OWL and SWRL combined with SPARQL querying facilities.

Section 4 describes a proof-of-concept implementation of the presented framework and reviews the requirements from Section 2 with respect to our contribution. In Section 5 we conclude this paper by giving an overview on open issues and an outlook on future research.

2 Use Case, Requirements and Related Work

For a better understanding and evaluating our research we present a use case describing an example request for a Custom Cloud Service. We use it to derive a set of required properties for our contribution. The section is concluded by an overview on the related literature.

2.1 Use Case

Consider a service request from a customer who wants to set up an online survey for a two months period. The customer has no preferences regarding the survey system, however would slightly prefer a Windows-based operating system over Linux. As he is quite acquainted with Oracle products, the survey system's underlying database is preferably also Oracle-based. The expected workload in form of concurrent users is unclear, however this is not considered a problem due to scalability of Cloud services.

In a typical scenario the provider wants to maximize profit, i.e. the difference between the offer price and the accruing costs. The challenge is to find the configuration that is in line with the customer preferences, thus having a high offer price, while simultaneously considering cost aspects.

2.2 Requirements

Based on the use case, we can now derive a set of requirements, defining and clarifying the goals of the desired solution. We identify five major properties that we find necessary to provide an adequate solution for finding the optimal service configuration in this case:

R 1 (Top Down Dependency Resolution). *Automatic resolution of all transitive dependencies between resource classes, starting from the top level functional requirement resources until no more unresolved dependencies exist. Thus deducting all variables for the Custom Cloud Service configuration problem.*

R 2 (Functional Requirements). *The functional requirements for the designated service should be describable by abstract or concrete resources (on different levels).*

R 3 (Customer Preferences). *The approach should be able to consider customer preferences regarding different configuration options.*

R 4 (Interoperability Check). *The interoperability/compatibility between resources has to be validated. For reducing the modeling overhead, this validation should be possible on both instance and higher abstraction levels.*

R 5 (Profit Optimization). *The profit maximizing Custom Cloud Service configuration has to be found. I.e. the configuration yielding the greatest difference between the achievable offer price for a configuration and the accruing costs on provider side.*

Additionally we require *correctness*, *completeness* and *consistency* for the designated decision support mechanism. However, as these properties are rather generic we do not consider them for our related literature review.

2.3 Related Work

For solving the technically and economically complex problem of deriving and optimizing configuration alternatives, we have to touch a broad spectrum of research areas, from Web service composition to techniques from operations research and constraint programming.

Berardi et al. [4] address the problem of automatic service composition by describing a service in terms of an execution tree, then making use of finite state machines to check for possible service compositions that match a requested behavior. Lécué et al. [11] present an AI planning-oriented approach using Semantics. Based on causal link matrices, the algorithm calculates a regression-based optimal service chain. Both [4] and [11] concentrate on input/output based matching of Web services, thus they are not suitable for infrastructure service compositions, where the interfaces are much more complex and cannot be described in terms of input and output. Another work from the Semantic Web context by Lamparter et al. [10] describes the matching process between requests and offers of Web services. This approach is a related example for the matching

of interoperability constraints (R 4), however it does not include dependency resolution.

Blau et al. [5] propose an ontology-based tool for planning and pricing of service mash-ups. The tool can be used to compose complex Web services from a set of known atomic services, which are stored in a domain specific ontology. Afterwards the complex service can be validated based on axioms and rules in the ontology.

Sabin et al. [15] present different constraint programming approaches for product configuration. Van Hoeve [7] describes optimization approaches for constraint satisfaction problems. In [9] an optimization framework combining constraint programming with a description logic is provided. All related to R 5.

We also want to mention two software tools that include a transitive dependency management, thus are mainly related to requirements 1 and 4. Advanced Packaging Tool (APT) [16] is a package management system to handle the installation and removal of software packages on Linux distributions. APT allows to automatically install further packages required by the desired software to avoid missing dependencies. The dependency management also includes compatibility checks, however only in form of a rather simple version level check. Another dependency manager is Apache Ivy [2]. Dependencies in Ivy are resolved transitively, i.e. you have to declare only direct dependencies, further dependencies of required resources are resolved automatically. Both approaches do not allow semantic annotations to allow more complex dependencies or interoperability constraints.

3 Custom Cloud Service Configuration

The contribution of our work consists of several elements in the context of configuring Custom Cloud Services. For a common understanding we first want to provide our own definition:

Definition 1 (Custom Cloud Service). *A Custom Cloud Service is a Cloud Computing service, composed by a set of software components and existing Cloud infrastructure resources, according to the abstract functional requirements and non-functional preferences of an individual customer or target customer group.*

In the following section we provide a formal definition on the functional requirements on Custom Cloud Services, tailored to the described use case. We then present an ontology model, that can be used to formally capture the knowledge about Cloud resources, their interdependencies and compatibility constraints as well as cost meta information. We also present an algorithm to build a dependency graph based on this knowledge, which can be seen as completely specified configuration problem. Lastly we show how the graph can be transformed into an integer program, allowing to find the optimal solution from a cost-benefit perspective. A sequence diagram to visualize the proposed interaction is depicted in Figure 1.

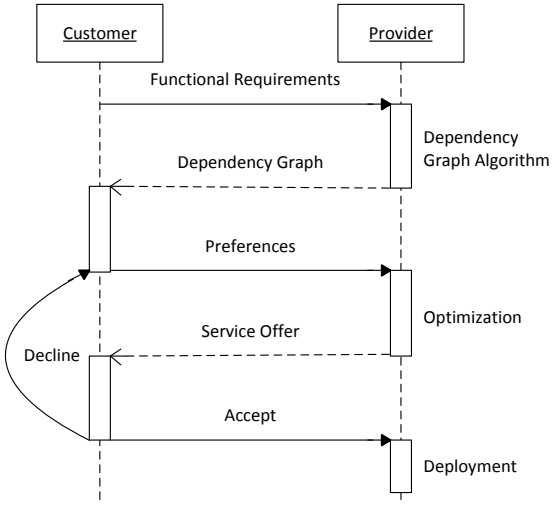

Fig. 1. Interaction Process

3.1 Functional Requirements

Functional requirements describe *what* is needed, i.e. the components required for a functioning service, whereas non-functional requirements describe *how* it is needed, i.e. the desired quality of the service. For the declarative description of functional requirements, we build on concepts taken from an *ontology* that contains background information in form of an abstract resource description layer (service ontology) and a domain-specific layer containing domain-dependent descriptions of available resources (domain ontology). We define functional requirements as follows.

Definition 2 (Functional Requirements). *For a background ontology O the functional requirements for deploying a service is described by the tuple $R = (\mathcal{C}, t)$, with $\mathcal{C} = \{C_1, \ldots, C_n\}$, a set of concepts in O, and t, the requested time period for the service.*

For illustration, the functional requirements of the presented use case would be the tuple $R = (\{OnlineSurvey\}, 1440h)$.

3.2 Knowledge Representation

For knowledge representation, we rely on the Web Ontology Language (OWL) [12] specification for several reasons. It has been established as the leading Semantic Web standard, is widely spread thus offering a large set of modeling tools and inference engines, is well documented and offers description logics (DL) expressiveness that fits well our anticipated description of services at class-level. In addition, OWL comes with a standardized, web-compliant serialization which ensures interoperability over the borders of single institutions.

For stating more complex compatibility constraints that go beyond the DL expressivity, we also make use of the Semantic Web Rule Language (SWRL) [8] in combination with OWL, while we restrict ourselves to DL-safe rules [14] as is a decidable fragment supported by OWL reasoners like Pellet [17] or KAON2 [13].

We make use of three different ontologies. A generic *service ontology*, which defines the fundamental concepts of our model. The actual knowledge on the known software and Cloud resources is modeled in the *domain ontology* and will differ for the various users or use cases. While the service ontology is a static model, the domain ontology has to be dynamic, i.e. it will need constant updates on the current infrastructure situation. The third ontology is used to store the results of the algorithm that derives all alternatives by dissolving the resources' dependencies. Both domain and *result ontology* import the fundamental concepts defined in the *service ontology*.

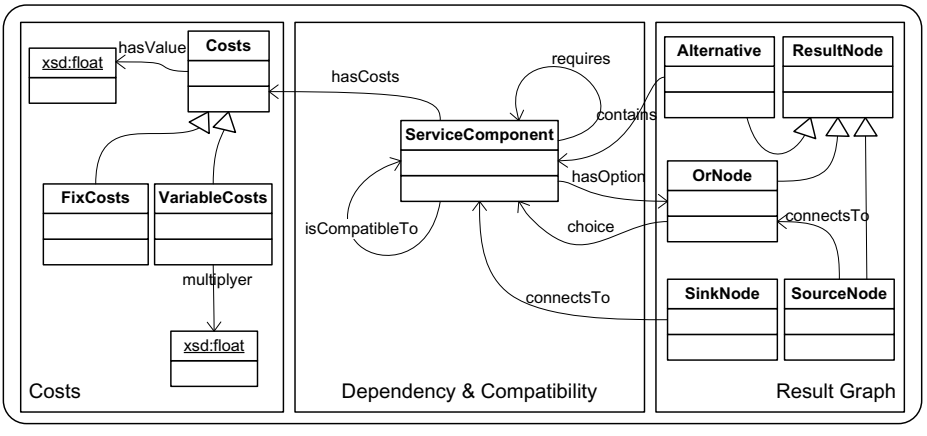

Fig. 2. Service Ontology

Service Ontology. The service ontology is partially depicted in Figure 2. For the reader's convenience we illustrate the ontology in UML notation where UML classes correspond to OWL concepts, UML associations to object properties, UML inheritance to sub-concept relations and UML objects to OWL instances. Basically, the service ontology provides concepts for three different aspects:

1. Service resources along with dependencies and compatibility information, needed to derive all valid service configuration alternatives
2. Cost meta information for evaluating these alternatives
3. Structure elements needed for an ontology representation of the dependency graph

The most fundamental concept for deriving all feasible deployment alternatives is the class *ServiceComponent* along with the corresponding object properties

requires and *isCompatibleTo*. The *requires* property is used to describe the functional dependency between two resource instances. In most cases, dependencies can and should be described in an abstract way at class-level. We can do this by including the dependency relation into the class axiom in conjunction with an object restriction in the form of an existential quantifier on the required class:

$$\begin{aligned} ComponentA \sqsubseteq\ & ServiceComponent \\ & \sqcap \exists requires.ComponentB \end{aligned}$$

Hereby we state that each resource of type *ComponentA* requires some resource of type *ComponentB*. As a more concrete example, we could state that every instance of the class Application requires at least some operating system:

$$\begin{aligned} Application \sqsubseteq\ & ServiceComponent \\ & \sqcap \exists requires.OS \end{aligned}$$

The compatibility can be asserted on instance level using the *isCompatibleTo* object property. That implies that there has to be one object relation between all possible combinations of interdependent resources. To reduce this modeling overhead, we propose the usage of SWRL rules, which allow us to state compatibility on class level:

$$isCompatibleTo(x,y) \leftarrow \quad \text{(L1.1)}$$
$$ComponentA(x) \quad \text{(L1.2)}$$
$$ComponentB(y) \quad \text{(L1.3)}$$

We can exploit the full expressiveness of DL-safe SWRL rules. E.g. to state that all versions of *MySQL* are compatible to all *Windows* versions except *Windows 95*, we include the following rule:

$$isCompatibleTo(x,y) \leftarrow \quad \text{(L2.1)}$$
$$MySQL(x) \quad \text{(L2.2)}$$
$$Windows(y) \quad \text{(L2.3)}$$
$$differentFrom(y,\text{'Windows95'}) \quad \text{(L2.4)}$$

For inclusion of cost information we distinguish between non-recurring *FixCosts* and recurring *VariableCosts*. The latter being more complex, as the total amount depends on another variable, which has to be defined in the context to serve as a multiplier. E.g. the overall usage fee for a cloud provider depends on the planned usage period for the service.

Domain Ontology. The domain ontology uses the concepts described in the preceding section to capture the knowledge about the Cloud service resources of

a certain domain of interest. This allows to easily use the same technology for many different contexts, just by loading a different domain ontology. In addition, knowledge can be combined by loading several domain ontologies, as long as this does not lead to inconsistencies.

Result Ontology. By resolving the transitive dependencies for the set of resources from the functional requirements, it is clear that we cannot add any knowledge, we can only make additional knowledge explicit, that is contained in the knowledge base implicitly.

For persisting the extended model, we also rely on an OWL ontology, such that it can be used for further reasoning tasks. The result ontology makes use of the concepts *SourceNode*, *SinkNode*, *OrNode* and *Alternative*, all defined in the service ontology. *SourceNode* and *SinkNode* are a helper nodes to have a distinct starting and ending points in the graph. They correspond to the source and sink nodes in a network. The *OrNode* is introduced to capture the branching whenever there is more than one compatible resource instance that fulfills the dependency requirement.

In the remainder of the paper we work with a more formal notation for the dependency graph. Note that both notations are semantically the same. An example graph is depicted in Figure 3.

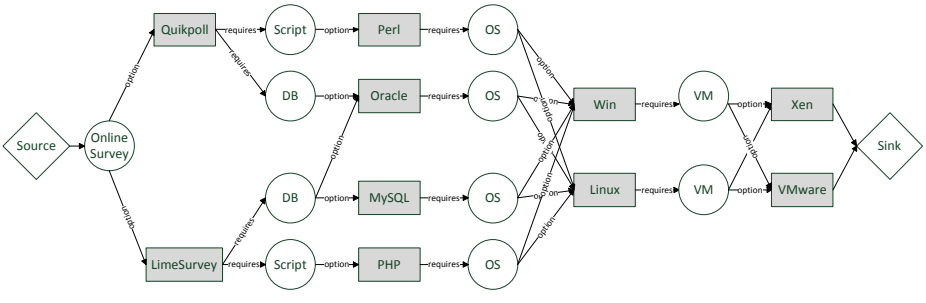

Fig. 3. Example Dependency Graph

Definition 3 (Dependency Graph). *For a background ontology O and a functional requirements tuple $R = (\mathcal{C}, t)$, the* dependency graph *$G = (V, E)$ is a directed, acyclic, labeled graph with vertices V and edges E and a labeling function $\mathcal{L}$, recursively defined as follows:*

- *$n_0 \in V$ is the source node of G*
- *$n_\infty \in V$ is the sink node of G*
- *there is a node $n_C \in V$ with label $\mathcal{L}(n_C) = C$ for each (atomic or nominal) class $C \in \mathcal{C}$ and an edge $(n_0, n_C) \in E$*
- *if $n \in V$ is a node with an atomic class label $\mathcal{L}(n) = A$ then there is a node n_o with label $\mathcal{L}(n_o) = \{o\}$ for each individual $o \in O$ with $O \models A(o)$, and an edge $e = (n, n_o)$ with label $\mathcal{L}(e) =$ or*

– *if $n \in V$ is a node with a nominal class label $\mathcal{L}(n) = \{o\}$ then there is a node n_C with label $\mathcal{L}(n_C) = C$ for each atomic or nominal class C with $O \models C(x)$ for all x such that $O \models requires(o, x)$ and $O \models isCompatibleTo(o, x)$, and an edge $e = (n, n_C)$ with label $\mathcal{L}(e) =$ and.*

3.3 Dependency Graph Algorithm

OWL reasoners typically construct models for answering standard reasoning tasks but do not expose them as such. Since we need to explicitly access such models as configuration alternatives at the instance-level, we chose an algorithmic solution, making use of OWL reasoning capabilities in between the construction of dependency graphs.

Algorithm 1. determineDependencies(R, O; G) – Initiate the construction of a dependency graph.

Require: a functional requirement tuple $R = (\mathcal{C}, F)$ and ontology O
Ensure: G contains the dependency graph for R

```
V := {n_0, n_∞}, V* := E := ∅
for all C ∈ 𝒞 do
   V := V ∪ {n_C}, ℒ(n_C) := C
   E := E ∪ {(n_0, n_C)}, ℒ((n_0, n_C)) = and
   for all o with O ⊨ C(o) do
      V := V ∪ {n_o}, ℒ(n_o) = {o}
      E := E ∪ {(n_C, n_o)}, ℒ((n_C, n_o)) = or
      deductServiceInfrastructure(O, o, G, V*)
   end for
end for
```

The procedure determineDependencies in Algorithm 1 initiates the construction of a dependency graph, starting from functional requirements R, and calls the procedure deductServiceInfrastructure in Algorithm 2, which recursively finds suitable service resource instances by following the object property *requires*.

In the procedure getRequiredClasses we invoke the reasoning engine with the following SPARQL query to find all implicitly stated dependencies, i.e. through a class axiom rather than explicitly on instance level:

```
SELECT ?sub ?t ?obj
WHERE {
   ?sub owl:sameAs dm:component .
   ?sub so:requires _:b0 .
   _:b0 rdf:type ?t.
   ?obj rdf:type ?t }
ORDER BY ?t
```

so hereby refers to the name space of the service ontology, *dm* to the name space of the domain ontology. The literal *_:b0* refers to a blank node, i.e. there

Algorithm 2. deductServiceInfrastructure(O, o; G, V^*) – Recursively construct a dependency graph for a given ontology and resource instance.

Require: an ontology O and a resource instance $o \in O$
Ensure: $G = (V, E)$ contains a dependency graph for o, V^* contains all resource instance nodes visited

```
V* := V* ∪ {n_o}
C := ∅, getRequiredClasses(o; C)
if C = ∅ then E := E ∪ {(n_o, n_∞)}
for all C ∈ C do
    V := V ∪ {n_C}, L(n_C) := C
    E := E ∪ {(n_o, n_C)}, L((n_o, n_C)) = and
    for all o' with O ⊨ C(o') and O ⊨ isCompatibleTo(o, o') do
        V := V ∪ {n_o'}, L(n_o') = {o'}
        E := E ∪ {(n_C, n_o')}, L((n_C, n_o')) = or
        if n_o' ∉ V* then deductServiceInfrastructure(O, o' , G, V*)
    end for
end for
```

do not exist two individuals for which we find the requires property, but based from the axiomatic knowledge we know there has to be at least one.

The query will answer us with a set of all types of these anonymous individuals. This has one disadvantage: we are only interested in the most specific class assertions. E.g. if it was stated that every application needs an operating system, the query would have the class *OS* in its result set, however also every superclass up to *Thing*.

Therefore, in a second step, we need to find out the most specific classes, i.e. all classes that have no subclasses also contained in the result set. This can be achieved by a simple algorithm which has a worst case runtime of $O(n^2)$ subsumption checks.

As there are redundant dependencies, which by themselves again might have further dependencies, we memorize the visited resources (V^*), as we do not need to resolve their dependencies more than once.

Further the algorithm remembers unfulfilled requirements and recursively traces them back, deleting unfeasible paths. We have not included these steps in the above printed pseudo algorithm, as they would only confuse the reader.

Constraint Satisfaction Problem In another formal representation of the graph (Figure 4), denoted as G^F, we can recognize the analogy to a constraint satisfaction problem [7], which is defined by a set of variables $\mathcal{X} = \{X_1, \ldots, X_n\}$, a set of domains $\mathcal{D} = \{D_{X_1}, \ldots, D_{X_n}\}$ defining the possible values for $\mathcal{X}$ and a set of constraints $\mathbb{C}$. In G^F the variables $\mathcal{X}$ are equal to vertices labeled with classes $C \in O$, and the domains D_{X_i} equal to vertices labeled with resource instances $o \in O$. Vertices, labeled with an identical resource class C, are subsumed by one variable X_C. The constraints in $\mathbb{C}$ can be derived from the edges denoting the interoperability between resources.

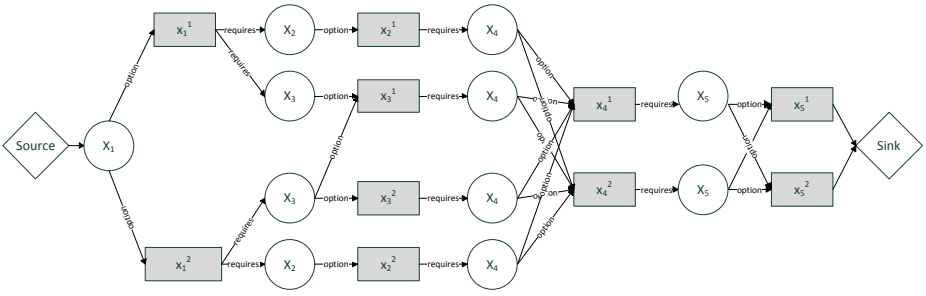

Fig. 4. Formal Dependency Graph

3.4 Preferences

As shown in Figure 1, after receiving the dependency graph, the customer can specify preferences. For quantifying the customer satisfaction for a certain configuration, we introduce the notion of a scoring function [3]. The scoring function maps the customer preferences on certain configuration choices to a real number in the interval $[0, 1]$. We achieve that by a weighted aggregation of the single preference values regarding the different configuration choices (cf. Section 3.5). For expressing non-compensating preferences, we define a set of additional restrictions $\mathcal{R}$ added to set of constraints $\mathbb{C}$.

Definition 4 (Preferences). *For a dependency graph G^F the customer preferences are described by the triplet $\mathcal{P} = (P, \Lambda, \mathcal{R})$, with $\mathcal{P} = \{P_1, \ldots, P_n\}$, a set of preference vectors, $\Lambda = \{\lambda_1, \ldots, \lambda_n\}$ a set of weights for each variable X_i in $\mathcal{X}$ with $\sum_i \lambda_i = 1$ and $\mathcal{R}$ a set of non-compensating restrictions.*

Example. For a variable X_{OS} representing the various operating system choices with $D_{X_{OS}} = \{Win, Linux\}$, the preferences are denoted by the vector $P_{OS} = (1, 0.8)^T$, expressing the slight preference for a Windows-based system, as described in our use case. Analogously, we would denote $P_{DB} = (0.5, 1)^T$ with $D_{X_{DB}} = \{MySQL, Oracle\}$. If both preference values are considered equally important, we would denote $\lambda_{OS} = \lambda_{DB} = 0.5$. A non-compensating restriction could be that the online survey has to be PHP-based, denoted by $\mathcal{R} = \{X_{Script} \overset{!}{=} PHP\}$.

3.5 Optimization

Having the dependency graph G^F and the customer preferences $\mathcal{P}$ we want to find the optimal configuration that will be offered to the customer. The optimal configuration can differ in various scenarios with different pricing schemes and potential additional constraints (like capacity restrictions). In this work we define the optimum as the configuration that yields the highest profit, i.e. the achieved price minus the accruing costs. Hereby we assume, that customer is sharing his

willingness to pay for a service perfectly fulfilling all his preferences, denoted by α. Extensions to this simple economic model are considered in ongoing research.

In a naive approach, we could try to iterate over all feasible configurations in G^F. One configuration is a sub graph, as the meaning of the edges can be interpreted as *and* and *or*. As a matter of fact, we can rewrite the dependency graph as Boolean formula.If we convert the Boolean formula (which is already in negation normal form) into its disjunctive normal form (DNF) by a step-wise replacement using de Morgan's laws, we exactly get an enumeration over all sets of resource instances that reflect the different configurations.

However, we are interested in the optimal configuration, thus iterating over all configurations might not be the best choice as it is very costly with respect to runtime. We therefore set up an integer program, which calculates the profit maximizing configuration. The variables from the constraint satisfaction problem are by modeled a vector of binary decision variables $X_i = (X_i^1, \ldots, X_i^m)$ for m different choices.

$$\begin{aligned} \underset{\mathcal{X}}{\text{maximize}} \quad & \alpha \cdot S(\mathcal{X}) - C(\mathcal{X}) \\ \text{subject to} \quad & \sum_{x_i^j \in X_i} x_i^j = 1 && \forall X_i \in \mathcal{X} \\ & X_i \cdot X_k \leq I_{ik} && \forall i, j : X_i \rightarrow_r X_k \\ & R \in \mathcal{R} && \text{constraints from } \mathcal{P} \end{aligned}$$

with

$$S(\mathcal{X}) = \sum_{X_i \in \mathcal{X}} \lambda_i \cdot X_i \cdot P_i$$

$$C(\mathcal{X}) = \sum_{X_i \in \mathcal{X}} C_i \cdot X_i$$

I_{ik} hereby denotes an interoperability matrix between X_i and X_k that can be derived from G^F. The cost function $C(\mathcal{X})$ merely is the sum over all costs for the chosen resources, which are stored in the ontology O. In case of variable costs, we multiply the cost value with the requested time period for the service t from R (cf. Section 3.1).

4 Evaluation

We evaluate our contribution qualitatively by having implemented a proof-of-concept prototype and comparing the presented approach to the requirements from Section 2.2.

4.1 Implementation

The implemented prototype can be executed using Java Web Start[1]. For using the prototype, a domain ontology (an example ontology is provided) has to be

[1] `http://research.steffenhaak.de/ServicePlanner/`

loaded, before one can add the set of resources $\mathcal{C}$. Eventually, the algorithms can be started by using the menu items *ResolveDependencies* and *Evaluate*. However, not all functionalities presented in this paper are integrated. We can derive all feasible configurations using the described DNF approach. The integer program has been implemented using CPLEX, thus not being part of the downloadable prototype. As reasoning engine we have chosen Pellet [17], as to our knowledge it is the only OWL DL reasoner that is capable of both SWRL rules and SPARQL queries that involve blank nodes.

4.2 Requirements Review

Taking a look back to the requirements derived from the use case, we have presented a framework to model knowledge about Cloud resource dependencies and compatibilities. Based on this knowledge base a set of functional requirements can be defined in from of classes from the ontology on arbitrary abstraction level. It is used to derive a dependency graph ensuring interoperability of all configurations. Based on the graph, the customer can define preferences for each variable, which are subsumed in a scoring function. Stated as constraint satisfaction problem, an integer program finds out the profit maximizing configuration, making use of the scoring function and the cost information stored in the ontology. The proposed approach therefore fulfills requirements R 1 to R 5.

5 Conclusion

In this paper, we propose an ontology-based optimization framework for finding the optimal Cloud service configuration based on a set of functional requirements, customer preferences and cost information. We do this by means of an OWL DL approach, combined with DL-safe SWRL rules and SPARQL querying facilities. We can model dependencies between resource classes of any abstraction level and use complex rules to ensure compatibility between resources. An integer program allows us to find the profit maximizing configuration. We provide a prototypical implementation as proof-of-concept.

We recognize several reasonable extensions and shortcomings to our approach. The economic model for the profit maximization is very simplistic. It is arguable that the customer is willing to give price his preferences and willingness to pay in a truthful manner. Further extensions from an economic perspective are the integration of capacity constraints and optimizing several concurrent requests. Ongoing research is dealing with both issues.

From the semantic perspective, a more complex cost and quality model would be beneficial. In addition, we plan to investigate on how the proposed domain ontology can be maintained collaboratively by incentivizing resource suppliers to contribute the necessary knowledge about their resources as interoperability and dependencies themselves.

We also recognize the need for a better evaluation, qualitatively, through relying on an industry use case, and quantitatively, by analyzing both the economic benefit of our solution and its computational complexity.

References

1. The Cloud Market EC2 Statistics (2010), http://thecloudmarket.com/stats
2. Apache. Apache Ivy (2010), http://ant.apache.org/ivy/
3. Asker, J., Cantillon, E.: Properties of Scoring Auctions. The RAND Journal of Economics 39(1), 69–85 (2008)
4. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic service composition based on behavioral descriptions. Int. J. of Cooperative Information Systems 14(4), 333–376 (2005)
5. Blau, B., Neumann, D., Weinhardt, C., Lamparter, S.: Planning and pricing of service mashups. In: 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services, pp. 19–26 (2008)
6. Gaimon, C., Singhal, V.: Flexibility and the choice of manufacturing facilities under short product life cycles. European Journal of Operational Research 60(2), 211–223 (1992)
7. van Hoeve, W.: Operations Research Techniques in Constraint Programming. Ph.D. thesis, Tepper School of Business (2005)
8. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A semantic web rule language combining OWL and RuleML. W3C Member submission 21 (2004)
9. Junker, U., Mailharro, D.: The logic of ilog (j) configurator: Combining constraint programming with a description logic. In: Proceedings of Workshop on Configuration, IJCAI, vol. 3, pp. 13–20. Citeseer (2003)
10. Lamparter, S., Ankolekar, A., Studer, R., Grimm, S.: Preference-based selection of highly configurable web services. In: Proceedings of the 16th international conference on World Wide Web, pp. 1013–1022. ACM Press, New York (2007)
11. Lécué, F., Léger, A.: A formal model for web service composition. In: Proceeding of the 2006 conference on Leading the Web in Concurrent Engineering, pp. 37–46. IOS Press, Amsterdam (2006)
12. McGuinness, D.L., Van Harmelen, F., et al.: OWL web ontology language overview. W3C recommendation 10, 2004–03 (2004)
13. Motik, B., Studer, R.: KAON2–A Scalable Reasoning Tool for the Semantic Web. In: Proceedings of the 2nd European Semantic Web Conference (ESWC 2005), Heraklion, Greece (2005)
14. Motik, B., Sattler, U., Studer, R.: Query Answering for OWL-DL with Rules. Journal of Web Semantics: Science, Services and Agents on the World Wide Web 3(1), 41–60 (2005)
15. Sabin, D., Freuder, E.: Configuration as composite constraint satisfaction. In: Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop, pp. 153–161 (1996)
16. Silva, G.: APT howto (2003), http://www.debian.org/doc/manuals/apt-howto/index.en.html
17. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical owl-dl reasoner. Web Semantics: Science, Services and Agents on the World Wide Web 5(2), 51–53 (2007)