A Semantic Recommendation Algorithm for the PaaSport Platform-as-a-Service Marketplace

[*]Nick Bassiliades[1,2], Moisis Symeonidis[1], Georgios Meditskos[1,3], Efstratios Kontopoulos[1,3], Panagiotis Gouvas[4], Ioannis Vlahavas[1,2]

nbassili@csd.auth.gr, mousisss@csd.auth.gr, gmeditsk@iti.gr, skontopo@iti.gr, pgouvas@ubitech.eu, vlahavas@csd.auth.gr

[1]School of Science & Technology, International Hellenic University, Thessaloniki, Greece

[2]Dept. of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece

[3]Information Technologies Institute, Centre of Research & Technology Hellas, Thessaloniki, Greece

[4]Ubitech Ltd., Athens, Greece

[*]Corresponding author

Address: Dept. of Informatics, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece

Email: nbassili@csd.auth.gr, Tel/Fax: +302310997913

Abstract

Platform as a service (PaaS) is one of the Cloud computing services that provides a computing platform in the Cloud, allowing customers to develop, run, and manage web applications without the complexity of building and maintaining the infrastructure. The primary disadvantage for an SME to enter the emerging PaaS market is the possibility of being locked in to a certain platform, mostly provided by the market's giants. The PaaSport project focuses on facilitating SMEs to deploy business applications on the best-matching Cloud PaaS offering and to seamlessly migrate these applications on demand, via a thin, non-intrusive Cloud-broker, in the form of a Cloud PaaS Marketplace. PaaSport enables PaaS provider SMEs to roll out semantically interoperable PaaS offerings, by annotating them using a unified PaaS semantic model that has been defined as an OWL ontology. In this paper we focus on the recommendation algorithm that has been developed on top of the ontology, for providing the application developer with recommendations about the best-matching Cloud PaaS offering. The algorithm consists of: a) a matchmaking part, where the functional parameters of the application are taken into account to rule out inconsistent offerings, and b) a ranking part, where the non-functional parameters of the application are considered to score and rank offerings. The algorithm is extensively evaluated showing linear scalability to the number of offerings and application requirements. Furthermore, it is extensible upon future semantic model extensions, because it is agnostic to domain specific concepts and parameters, using SPARQL template queries.

Keywords

Cloud computing; Platform-as-a-Service; Semantic Interoperability; Platform offering; Cloud application; Recommendation; Semantic matchmaking; Ranking

1 Introduction

Platform as a service (PaaS) is a cloud computing model that delivers applications over the Internet. A cloud PaaS provider delivers hardware and software tools, usually those needed for application development, to its users as a service. As a result, PaaS frees users from having to install in-house hardware and software to develop or run a new application. It, thus, helps save on the cost incurred for buying and managing the underlying hardware and software. The PaaS model minimizes the incremental cost required for scaling the system with growth in the service usage, while allowing for resource sharing, reuse, life-cycle management, and automated deployment. For these benefits, PaaS is preferred over other solutions for application and service development.

There are many market forecasts, e.g. (Carvalho, Mahowald, McGrath, Fleming, & Hilwa, 2015; Columbus, 2013), indicating that PaaS has a very positive economic outlook for the Cloud market. Although giant vendors occupy this emerging space, including Microsoft, Amazon, Google, and Salesforce.com, many startups and SMEs try to enter the market. The major problem is that the big vendors battle for dominance and they are reluctant to agree on widely accepted standards promoting their own, mutually incompatible Cloud standards and formats (Gagliardi & Muscella, 2010). This dominance increases the lock-in of customers in a single Cloud platform, preventing the *portability* of data or software created by them. But even if portability is supported, the high complexity and the additional switching costs discourage users from doing so (Androcec, Vrcek, & Kungas, 2015). Cloud specialists argue that in the years to come both large and small Cloud PaaS providers will grow through partnerships (Gardner, 2010). The formation of partnerships and federations between heterogeneous Cloud PaaS Providers involves *interoperability*. Cloud community and the EC (European Commission, 2012) have realized the significance of interoperability and has initiated several approaches to tackle this challenging issue. The first efforts to explore interoperability in PaaS are also well on track, e.g. CAMP (OASIS CAMP TC, 2014).

In order to raise the barriers that prevent the small-medium Cloud PaaS vendors (existing and potential ones) and software SMEs from entering the PaaS market, companies developing applications should be able to choose between different Cloud PaaS offerings, e.g. selecting the most reliable, the most well-reputed, the most cost-efficient or simply the one that meets their technical requirements, and should also be able to switch easily and transparently between Cloud providers whenever needed, e.g. when an SLA is breached or when the cost is too high, without setting data and applications at risk, e.g. loss of data or inability to run the application on a different platform. Moreover, they should be able to compare Cloud offerings with different characteristics, such as resource, pricing or Quality of Service (QoS) model, and to choose the one that best matches their computing needs of their services and applications (Borenstein & Blake, 2011).

Having the above in mind, the PaaSport project (PaaSPort FP7 project, 2016) aims to resolve the application and data portability issues of the Cloud PaaS market through a flexible and efficient deployment and migration approach. These include, but are not limited to: image conversion to be suitable for target hypervisor, compression to aid, speed of transfer, image encryption, secure protocols, QoS guarantees, trust issues and cost sharing models. To this end, PaaSport combines Cloud PaaS technologies with lightweight semantics in order to specify and deliver a thin, non-intrusive Cloud-broker (in the form of a Cloud PaaS Marketplace), to implement the enabling tools and technologies, and to deploy fully operational prototypes and large-scale demonstrators.

PaaSport's scope is Cloud vendors (in particular SMEs) to be able to roll out semantically interoperable PaaS offerings leveraging their competitive advantage, the quality of service and value delivered to their customers, and improving their outreach to potential customers, particularly the software industry. Semantically Interoperable PaaS Solutions constitute Cloud PaaS systems and offerings with the ability to overcome the semantic incompatibilities and communicate, referring to the ability of applications and their data to seamlessly be deployed on and/or migrated between Cloud PaaS offerings that are using the same technological background but different data models and APIs. PaaSport will also enable Software SMEs to deploy business applications on the best-matching Cloud PaaS and to seamlessly migrate these applications on demand. PaaSport contributes to aligning and interconnecting heterogeneous PaaS offerings, overcoming the vendor lock-in problem and lowering switching costs.

To tackle the above, the PaaSport project has, among others, a) developed an open, generic, thin Cloud-broker for a marketplace of semantically-interconnected, interoperable PaaS offerings, resolving the main PaaS interoperability and application portability issues, b) defined a unified semantic model for PaaS offerings and business applications, c) developed a Unified PaaS API that allows the deployment and migration of business applications transparently to the user and independent of the specificities of a PaaS offering, d) defined a unified service-level agreement model addressing the complex characteristics and dynamic environment of the Cloud PaaS marketplace, and e) defined a Cloud PaaS offerings discovery, short-listing and recommendation algorithm for providing the user with the best-matching PaaS offering. In this paper, we focus on few of the above; we initially briefly present the marketplace architecture (a) and the unified semantic model (b) and we extensively report on the recommendation algorithm (e). Furthermore, we also present how the algorithm has been integrated into the Persistence Layer of the PaaSport marketplace.

Specifically, in order to support the PaaSport marketplace, a unified semantic model has been defined as an OWL ontology, for representing the necessary concepts and attributes for the definition of PaaS offerings and the business applications to be deployed through the proposed Cloud Marketplace (PaaSport Consortium, 2014c). The PaaSport ontology has been defined as an extension of the DOLCE+DnS ontology design

pattern (DUL) (Gangemi & Mika, 2003), offering extensibility, since both PaaS concepts and parameters are defined as classes, so extending the ontology with new concepts requires just to extend class hierarchies without adding ontology properties.

On top of the ontology, a semantic matchmaking and ranking algorithm has been developed for providing the application developer recommendations about the best-matching Cloud PaaS offering, using SPARQL queries for retrieving relevant data from the semantic repository. The algorithm consists of two parts: a) the matchmaking part, where the functional parameters of the application profile are taken into account to rule out inconsistent offerings, and b) the ranking part, where the non-functional parameters of the application profile are taken into account to score offerings and rank them according to this score. Due to the fact that application requirements and PaaS offering share the same vocabulary for PaaS concepts and parameters, the recommendation algorithm seamlessly matches application requirements to PaaS offerings both syntactically and semantically. The matchmaking and recommendation algorithm has complexity that is linear to the number of instances and the number of concepts and parameters. Furthermore, the algorithm is extensible because it is agnostic to domain specific concepts and parameters.

In the rest of this paper, we initially review related work in Section 2 on matchmaking / recommendation algorithms for cloud / service computing. We then briefly present the PaaSport marketplace architecture in Section 3, including the PaaSport Semantic Models, and we briefly discuss the requirements for the PaaSport recommendation algorithm. In Section 4, we present in detail the recommendation algorithm, its implementation using SPARQL query templates, as well as how it has been integrated with the persistence layer of the PaaSport marketplace. Finally, Section 5 presents the evaluation of the scalability of the recommendation algorithm, whereas Section 6 concludes the paper with a discussion on future work.

2 Related Work

Since very few related work can be found on the exact topic of matchmaking cloud applications to PaaS offerings, this section presents indicative examples of the state-of-the-art in various matchmaking algorithms, focusing mainly on (semantic) web service matchmaking algorithms, which are the most popular cases of matchmaking, highlighting the underlying principles and techniques. Existing matchmaking techniques can be classified as logic-based, syntactic or text-similarity-based, while structural similarity and learning are less common. Logic-based techniques range from straightforward series of few class-relationship rules to large lists of semantic conditions through ontologies. Text-similarity mostly uses textbook algorithms from the field of Information Retrieval.

SAWSDL-MX (Klusch & Kapahnke, 2008) provides all the standard matching strategies, namely logic-based, syntactic (text-similarity) and hybrid (logic-based and syntactic similarity). The strategies target service input, output and their underlying components (e.g. ComplexType), trying to find a match between a requested service (i.e. query) and all services offered in a set. Each offered service's operation is matched with every requested operation and rated with the maximum observed match. An offered service's overall rating is the worst (minimum) rating of all requested operations. However, SAWSDL-TC contains single-operation services only. Hence, operation-match rating is equal to the overall service-match rating. SAWSDL-MX2 (Klusch, Kapahnke, & Zinnikus, 2009), in addition to logic-based and text-similarity, measures structural similarity between WSDL (Web Services Description Language) file schema information (e.g. element names, data types and structural properties), using WSDL-Analyzer. It also introduces an adaptive, learning layer where SVM (Support Vector Machine) training vectors consist of values for logic-based, textual and structural criteria. Finally, iSeM (Klusch & Kapahnke, 2010) is an evolution of the MX series. In principle, it applies SVM learning for the weighted aggregation of underlying algorithm rankings. The learning vectors are an extended version of the ones in –MX2, containing logic, structural and text similarity in similar fashion to –MX algorithms. However, approximate logic matching was added, which captures more matches than the existing one, using more relaxed criteria for subsumption.

XAM4SWS is a common framework, from which two algorithms were derived, LOG4SWS and COV4SWS (Lampe & Schulte, 2012). Both algorithms perform operation-centric matching, targeting service interfaces, operations and I/O. LOG4SWS performs logic-based matching, in an –MX fashion, mapping ratings to numbers using linear regression. Meanwhile, COV4SWS rating measures are inspired by the field of semantic relatedness. It then performs regression to find weights for the aggregation of ratings (from underlying service elements to an overall service rating).

iMatcher (Wei, T. Wang, J. Wang, & Bernstein, 2011) integrates interesting variations of well-known strategies. The first strategy includes three sub-strategies. It performs text-similarity aimed at either the WSDL service name field, service description field or semantic annotations. The second strategy selects the maximum rating between two sub-strategies. The first is a hybrid variant, where the logic-based part rates inputs and outputs of operations with 1 if the requested concept is a parent of the offered concept. If logic-based matching fails, syntactic matching is performed. The second sub-strategy examines the distance of two concepts originating from different ontologies using ontology alignment similarity. On top of that, iMatcher also implements an Adaptive Matching method. The user selects multiple strategies, the results of which form vectors of the training set. Learning is performed by selecting an algorithm from the Weka library (Hall et al., 2009).

Most of the logic-based / ontology-based works on semantic web service matching have been more or less influenced by the seminal work by (Paolucci, Kawamura, Payne, & Sycara, 2002), which proposed a semantic matchmaking algorithm of web service capabilities based on the use of DAML-S ontology. In this algorithm a match between an advertisement and a service request, consists of a similarity degree obtained by matching all the input / output parameters of the request with those of the advertisement. A parameter from the provider matches with one from the request profile if there is a conceptual relation in the ontology. Four different measures are defined with a matching degree organized along the scale *exact* > *plugin* > *subsumed* > *fail*.

Later on the above algorithm has been extended by various researchers, such as (Meditskos & Bassiliades, 2010), (Zapater, Escrivá, García, & Durá, 2015), introducing new similarity measures (e.g. sibling), to improve recall, or using pre-process filtering steps based on web service categorization, to reduce the set of potentially useful web services, improving the scalability of the semantic matching algorithm. Furthermore, the Skyline system (Skoutas, Sacharidis, Simitsis, & Sellis, 2008) used an interesting strategy worth mentioning. Its target components are IOPEs, namely Inputs together with Preconditions and Outputs with Effects. The strategy is logic-based and classifies cases, such as *Exact*, *Direct_Subclass*, *Subclass*, *Direct_Superclass*, *Superclass*, *Sibling* and *Fail*. To find the optimal trade-off of input versus output significance, the homonymous algorithm, Skyline, is used. Hence, services with an optimal combination of input and output ratings are returned.

Compared to all the above web service matchmaking algorithms, our algorithm uses a more straightforward semantic matchmaking degree, based only on exact and plugin matching, which are both considered equivalent. However, we offer a much richer similarity measure on non-functional or QoS parameters, which allows for better ranking of the advertisements.

In the following we also discuss the relevance of various European research projects that deal with matchmaking issues for cloud computing. Cloud4SOA (Cloud4SOA FP7 project, 2012; Kamateri et al., 2013) is a scalable solution to semantically interconnect heterogeneous PaaS offerings across different cloud providers that share the same technology. The design of Cloud4SOA comprises of a set of interlinked software components and models to provide developers and platform providers with a number of core capabilities: matchmaking, management, monitoring and migration of applications. As far as the semantic matchmaking is concerned, the objective is to resolve semantic conflicts between diverse PaaS offerings. To this end, it aligns the application model requirements and the PaaS offerings even if they are expressed in different terms, resolving semantic conflicts and allowing the matching of concepts between different PaaS providers. The outcome is a list of PaaS offerings that satisfy developer's needs, ranked according to the number of

satisfied user preferences. The matchmaking algorithm of Cloud4SOA returns a ranked list of PaaS Offerings that satisfy the requirements described by the Application Instance (Bosi, Ravagli, Laudizio, Porwol, & Zeginis, 2012). The ranking is calculated as a percentage of the Application's preferences satisfied by the PaaS Instance against all the Application's preferences.

In PaaSport the focus has been given on defining a lightweight semantic matchmaking algorithm (see Section 4). To this end, we have defined a SPARQL-centric algorithm, using SPARQL queries to:

a. retrieve data from the underlying semantic repository, and
b. to implement the scoring functions.

In addition, the semantic representation of offerings, application models and SLAs follow the Descriptions and Situations pattern of DUL (see Section 3.1), in contrast to the majority of existing frameworks that use the OWL-S and SAWSDL models. Although OWL-S and SAWSDL provide a quite rich set of knowledge constructs for modelling various aspects of services, they have limited expressivity, since the modelling is based on defining instances instead of concepts (e.g. instances of the owls:Profile class). By following the DnS pattern, services are represented as complex concepts, mapping them in domain ontologies as a whole, and the matchmaking examines the subsumption relationships.

Our matchmaking and ranking algorithm has a similar workflow to the corresponding algorithm of Cloud4SOA; however, there are many differences in the implementations of the workflow. First of all, we do not use a single SPARQL query for checking all the functional parameters, as explained and justified in section 4. Secondly, our scoring function is much more elaborate than that of Cloud4SOA, including user guidance on the weight of each non-functional parameter, as well as a guide on how to interpret differences. Finally, since our semantic model offers a rich selection of data types, including measurement units, we offer far more complicated SPARQL queries to calculate conformance and scoring.

The *MODAClouds* project (MODAClouds FP7 project, 2016) provides: i) a Decision Support System (DDS); ii) an Integrated Development Environment (IDE); iii) a run-time environment for the high-level design, early prototyping, semi-automatic code generation, and; iv) automatic deployment of applications on Multi-Clouds with guaranteed QoS. In particular, MODAClouds uses a Model-Driven Engineering approach for Clouds for semi-automatic code deployment using decision support systems on multiple Cloud providers hiding the proprietary technology stack. Target environments for the MODAClouds framework cover IaaS, PaaS and SaaS solutions spanning across all abstraction layers, supporting public, private, and hybrid Clouds. MODAClouds targets Cloud application developers and administrators while PaaSport targets Software DevOps Engineers and PaaS providers. However, both projects aim at delivering methods for the platform-neutral description of cloud services, as well as capabilities for the run-time monitoring. The decision support aspects of MODAClouds are based on a recommendation system that provides the end user

with an overview of cloud services from multiple providers meeting their requirements. Its key features include:

- multiple stakeholder participation in specification of requirements,
- risk analysis based approach to generate cloud service characteristics meeting the requirements,
- a three dimensional (risk, cost and quality) assessment of offered cloud services to allow users choose based on the prime properties of interest to them,
- ensuring the multi-clouds related characteristics are accounted for in set of services obtained for different entities to be cloudified,
- easy visualization and comprehension of the elaborate process of decision making.

The decision support aspects of MODAClouds are similar to that of the recommendation layer of PaaSport. The matchmaking technique used in MODAClouds (Gupta et al., 2015) is quite different from that of PaaSport, based on the risk assessment method of (Lund, Solhaug, & Stolen, 2011), whereas PaaSPort is based on a multi-criteria approach. Both systems end up with a shortlist of recommended services; the difference being that PaaSport is aimed towards a single cloud provider, whereas MODAClouds can end up with multiple vendors for different services. Furthermore, our approach is much simpler for the DevOps engineer and the PaaS provider regarding data gathering for the application requirements and the PaaS offering characteristics. Finally, we extensively evaluate the PaaSport recommendation algorithm for scalability, whereas the risk-based decision making tool of MODAClouds has been evaluated in a much smaller scale, regarding scalability.

The *4CaaSt* project (4CaaSt FP7 project, 2013) (developed a PaaS framework for enabling flexible definition, marketing, deployment and management of Cloud-based services and applications. The project introduced the concept of blueprint, a technical description of an application or a service that decouples the various dependencies along the Cloud layers. Using these 4CaaSt blueprints, application providers can choose across different platform layers and services to run their applications, including different infrastructure, middleware, and applications components/services. Once the selection is done, 4CaaSt generates automatically the deployment designs and automatically provisions the necessary resources required for the deployment.

The offering selection mechanism of 4CaaSt (Andrikopoulos, Zhe, & Leymann, 2013) is based on a decision support system that incorporates both offering matching and cost calculation. The offering matching algorithm is based on a fixed offering model that supports only a subset of the offering concepts and parameters that PaaSport supports; furthermore, PaaSport is an extensible semantic model based on an OWL ontology that extends the DUL upper ontology, without the need to reconfigure neither the database schema nor the ontology itself. The matchmaking algorithm itself considers every required QoS parameter as a functional

parameter, whereas in PaaSport we follow a more flexible approach, leaving this decision to the DevOps Engineer. The ranking scheme of 4CaaSt is not as elaborate as PaaSport's and ranking is based always on a single parameter, not a combination of all parameters as in PaaSport. Finally, there is no evidence that 4CaaSt has been evaluated for scalability as PaaSport.

The main objective of the *PaaSage* project (PaaSage FP7 project, 2016) was to assist the developer with difficult deployment scenarios through autonomic cloud deployment, including orchestration of simultaneous deployment of various application parts to many different Clouds. The scope of PaaSage is to extend the application model with platform annotations and user's goals and preference using a domain specific language model, which is then transformed by PaaSage to a deployed application in one or more Clouds. To support the developer, PaaSage needs also models of the features of the available Cloud platforms, and goals and preferences to be satisfied by the deployment like response times or deployment cost budgets. The application's model is used to derive a specific deployment configuration satisfying all the constraints and goals for the deployment set by the user. This implies selecting one or more Cloud providers and generating the necessary deployment scripts. These scripts are then passed to the PaaSage execution ware responsible for instantiating the different parts of the application on the selected Cloud providers and monitor a set of defined metrics in order to make autonomous scalability decisions within the boundaries of the deployment configuration.

In order to select the appropriate cloud environments to deploy the application, PaaSage uses a Software Product Lines-based platform (SALOON) (Quinton, Romero, & Duchien, 2016), which relies on feature models combined with a domain model (ontology) used to select among cloud environments a well-suited one. Feature models are used to model a specific cloud environment, from a single cloud vendor. Then, all these feature models are manually mapped by domain experts to the single domain model (ontology) of PaaSage. Once the application developer defines the application requirements, SALOON searches for a cloud environment that fulfill these requirements. This search is automated by relying on feature models with attributes and cardinalities. Once a configuration is defined for each feature model using the mapping relationships, SALOON provides a reasoning engine to (i) check whether the configuration is valid or not and (ii) estimate the cost of such a configuration. Once FMs are configured, SALOON translates the FM into a Constraint Satisfaction Problem (CSP) and solves the latter using a CSP Solver to check whether the configuration is valid.

Compared to PaaSport, PaaSage / SALOON do not provide ranking of multiple cloud providers, but rather they are used to check whether a given cloud provider is compliant with a given application and to guide the developer in order to configure the application in a possibly multi-cloud setting. SALOON supports

stakeholders while configuring the selected cloud environment in a consistent way, and automates the deployment of such configurations through the generation of executable configuration scripts. This paper also reports on some experiments showing that using SALOON significantly reduces time to configure a cloud environment compared to a manual approach and provides a reliable way to find a correct and suitable configuration. Moreover, our empirical evaluation shows that our approach is effective and scalable to properly deal with a significant number of cloud environments. Furthermore, the performance of SALOON was proved to be scalable and affordable to reasonable settings, but superlinear, compared to PaaSport that its performance is linear.

3 Overview of the PaaSport Marketplace

The PaaSport Marketplace is an infrastructure that facilitates the publication and advertisement of semantically-interconnected available Cloud PaaS offerings, the identification and recommendation of the best-matching PaaS offering, and the seamless business application deployment and migration. The architecture of the PaaSport Marketplace (**Figure 1**) constitutes a thin, non-intrusive broker that mediates between competing or even collaborating PaaS offerings (PaaSport Consortium, 2014b). It relies on open standards and introduces a scalable, reusable, modular, extendable and transferable approach for facilitating the deployment and execution of resource intensive business services on top of semantically-enhanced Cloud PaaS offerings. Notice that the scope of PaaSport mainly involves a) PaaS offerings that originally support high degree of portability, which is classified to category 1 according to (Walraven, Truyen, & Joosen, 2014), and b) applications that are fully migrated to the cloud, either as complete software stacks in a VM or as a composition of services running on the Cloud, which are classified to types III and IV according to (Andrikopoulos, Binz, Leymann, & Strauch, 2013). PaaSport comprises of the following five artefacts:

- Adaptive Front-ends that support seamless interaction between the users and the PaaSport functionalities, through a set of configurable utilities that are adapted to the user's context;
- PaaSport Semantic Models that serve as the conceptual and modelling pillars of the marketplace infrastructure, for the annotation of registered PaaS offerings and deployed applications profiles;
- PaaS Offering Recommendation Layer that implements the core functionalities offered by the PaaSport Marketplace Infrastructure, such as PaaS offering discovery, recommendation and rating;
- Monitoring and SLA Enforcement Layer that realizes the monitoring of the deployed business applications and the corresponding Service Level agreement;
- Persistence, Execution and Coordination Layer that puts in place the technical infrastructure, e.g. repositories, on top of which the PaaSport marketplace is built.

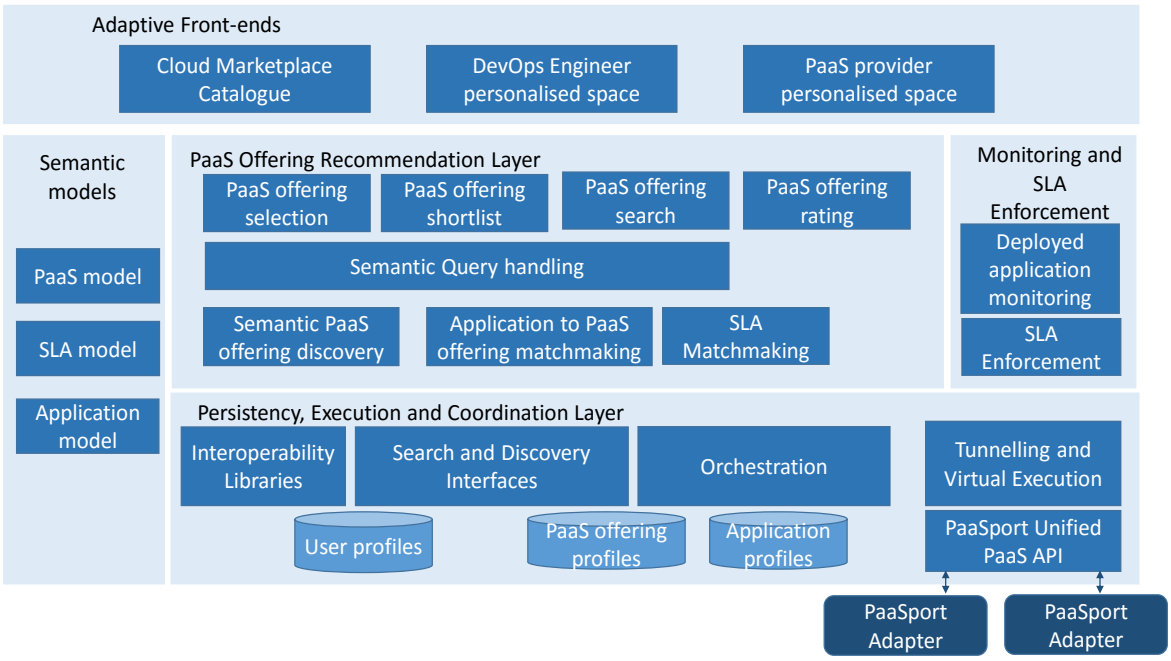

Figure 1. High-level view of the PaaSport Cloud-broker Architecture.

Our focus in this paper is to present thoroughly how semantics are used in the PaaSport marketplace in order to implement effective semantic matchmaking between the PaaS offerings and an application requirement profile, combining the Semantic Models, the Offering Recommendation layer and the persistence layer. The PaaSport Semantic Models are the conceptual and modelling pillars of the marketplace infrastructure and they are used in order to provide a semantic annotation means for the registered PaaS offerings and the deployed applications profiles (see Section 3.1).

Concerning the Offering Recommendation layer, the Semantic Models bridge the gap between business application requirements and PaaS offerings capabilities, thus, facilitating the matchmaking and the identification of the specific PaaS that fulfills the business and technical requirements of a particular application. This is achieved by using a common conceptual framework for describing both the platform offering and the application model, so that application requests can be matched conceptually, structurally and quantitatively to platform offerings, as explained in section 4.

Concerning the Persistence layer, the database schema follows exactly the conceptual model of the ontologies, in order to avoid syntactic/semantic mismatch between tables/concepts and attributes/properties when the Offering Recommendation layer retrieves PaaS offerings from the database, based on the Semantic Models. This is achieved by mapping the PaaS Offering profiles stored in the database onto the Semantic Model layer (RDF graph) using a relational-to-ontology mapping tool (see section 4.4).

The main stakeholders of the PaaSport Marketplace, identified in PaaSport project deliverable D1.1 (PaaSport Consortium, 2014a), that concern the semantic models and the recommendation algorithm are:

- **DevOps Engineer**: A solution architect seeking an optimal PaaS platform to develop, deploy or migrate to a complex cloud application. The most important optimization criteria for the search and decision-making process are the technical requirements for the platform.
- **PaaS Provider**: An enterprise whose business model includes the delivery and operation of one or more PaaS solutions. A PaaS provider defines the technical aspects, the pricing models, reference values for quality of service parameters, and terms and conditions that apply to their offerings.

PaaS providers supply the Cloud-based application developers with the available PaaS offerings. The DevOps Engineers build applications that will be deployed and executed on PaaS offerings (platform services). The DevOps Engineers search for PaaS offerings that satisfy their applications' requirements. After a successful negotiation with a PaaS provider the applications are deployed on the PaaS offering by the service consumer for whom the DevOps Engineers built the applications. An application can vary from a relational DBMS or a lightweight Web application to a heavy software system, e.g. an ERP or a CRM.

The recommendation algorithm is mainly used by the DevOps Engineer for searching PaaS offerings that meet the functional and non-functional requirements an application imposes on its platform. The DevOps Engineer engages this functionality when he/she wants to search for a platform to deploy its application (through the PaaSport broker), or to migrate its application, in case of SLA violation. The DevOps Engineer (in the full-fledged PaaSport platform) is using a GUI to enter the requirements for his/her application (application profile). There are two types of requirements: a) functional requirements, which are absolutely needed by the application and cannot be negotiated (e.g. the application needs a MySQL DBMS instance), and b) non-functional requirements, which are preferable but negotiable (e.g. storage capacity 10GB).

After the DevOps Engineer enters the above requirements, the presented algorithm will run on the PaaS Offering Recommendation Layer and it will return to the DevOps Engineer a list of PaaS platforms that meet his/her functional requirements, along with a score, based on his/her non-functional requirements. The list will be sorted in descending order of score (highest first).

The most important requirements for the PaaSport Recommendation Algorithm are:

- *The recommendation algorithm should be efficient and scalable*. As we show in Sections 4.1 and 5, the recommendation algorithm has complexity that is linear to the number of PaaS offerings and application requirements.
- *The recommendation algorithm should be easily extensible*, i.e. when the semantic model is extended with new offering instances, concepts and parameters, the algorithm should be minimally

changed. Ideally the algorithm should not change at all. As we show in Section 4.1, the recommendation algorithm is extensible because it is agnostic to domain specific concepts and parameters.

3.1 Overview of PaaSport Semantic Models

In this sub-section, we give a brief overview of the PaaSport Semantic Models to help comprehension of the recommendation algorithm fully described in the next section (Section 4), since the algorithm runs on top of the PaaSport ontology, using SPARQL queries for retrieving relevant data from the semantic repository. More details about the ontology development can be found at (PaaSport Consortium, 2014c). The PaaSport ontology is divided into three models:

1. **Offering Model**: facilitates the semantic annotation of the available PaaS offerings in terms of functionalities, resources and business characteristics offered.
2. **Application model**: enables the semantic annotation of the application to be deployed at the PaaSport marketplace, in terms of functionalities, resource and requirements in business characteristics.
3. **SLA Model**: allows the annotation of the service-level agreements provided and supported by the registered Cloud PaaS offerings. Since it is not required for the algorithm, it is not discussed further.

Notice that the Offering and Application models are not very different. Only the top-level classes are different, since Offerings and Applications stand conceptually for two very different types of entities: cloud platforms and cloud applications, respectively. However, both use the same vocabulary for describing services (PaaS concepts) and their parameters, as shown in **Figure 2**. This allows the recommendation algorithm to seamlessly match application requirements to PaaS offerings both syntactically and semantically.

In the PaaSport project we have decided to use the DOLCE+DnS Ultralite (DUL) ontology, which is a simplification and an improvement of some parts of DOLCE Lite-Plus library and Descriptions and Situations ontology (Gangemi & Mika, 2003). The main reasons we have followed this approach are interoperability and extensibility. The use of DUL (i.e. an upper ontology) ensures better semantic interoperability with other similar projects and research efforts. Furthermore, the DUL ontology is very easy to be extended by adding e.g. new concepts and parameters related to PaaSport offerings and applications. Properties (i.e. parameters) are defined as new classes and not as OWL properties. In this way, the introduction of new properties/parameters does not require the disturbance of the schema of existing tables of the persistence layer of the PaaSport broker, which is built using a relational database, but merely the introduction of new tables. Furthermore, this representation of properties also favors the generality and extensibility of the recommendation algorithm between application requirements and platform offerings. This type of extensibility is analyzed in section 4.

The core modeling pattern of DnS allows the representation of the following conceptualizations (**Figure 2**):

- **Situations**. A situation defines a set of domain entities that are involved in a specific pattern instantiation (isSettingFor property) and they are interpreted on the basis of a description (satisfies property). Each situation is also correlated with one user/agent (dul:includesAgent property).
- **Descriptions**. An activity description serves as the descriptive context of a situation, defining the concepts (defines property) that classify the domain entities of a specific pattern instantiation, creating views on situations.
- **Concepts**. The DUL concepts classify domain entities describing the way they should be interpreted in a particular situation. Each concept may refer to one or more parameters, allowing the enrichment of concepts with additional descriptive context.

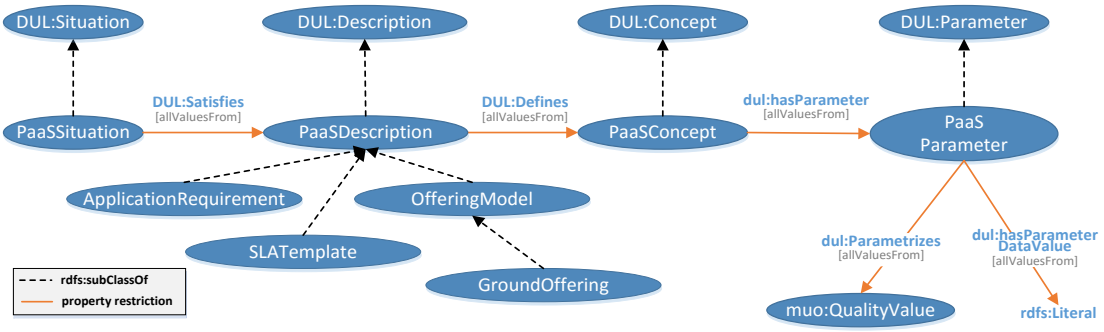

Figure 2. Abstract PaaSport design pattern.

The PaasPort semantic models have been defined as extensions to the core DnS pattern (**Figure 3**, **Figure 4**). More specifically, the *situation* concept is used as a container for defining the higher level conceptualizations of the PaaSport domain, such as PaaS *offerings*, *application* models and SLAs. The extensibility of our model depends on the parameters' definition. The specialized situations (offering, application, SLA agreement) are further correlated with specializations of the *description* concept, allowing the correlation of the situations with additional descriptive context, namely a set of one or more *concepts*, which define matchmaking dimensions for PaaS. A concept (for example QoS) could have one or more *parameters* (e.g. latency). Every parameter has a *value* (through dul:hasParameterDataValue property) and a "quality" (through dul:parametrizes property), which is the "physical" or "logical" dimension of the parameter (e.g. storage, duration, etc) and it is usually (not always) accompanied by measurement units. The abstract pattern associating DUL and PaaSport models is presented in Figure 2 and constitutes a rich, dynamic and flexible modelling pattern able to fully address the PaaS domain. Notice that PaaS concepts and parameters are defined once and are used both by the offering and application models, since the restrictions on the dul:defines property are inherited by all subclasses of PaaSDescription.

The Offering Model provides the vocabulary for semantically describing PaaS offerings. It contains all available characteristics of a PaaS platform, such as technical, performance-related, geographical etc., and it can be used for describing common attributes of a specific instantiation, such as programming language, databases, server etc. The extension of the abstract PaaSport model is presented in **Figure 3**.

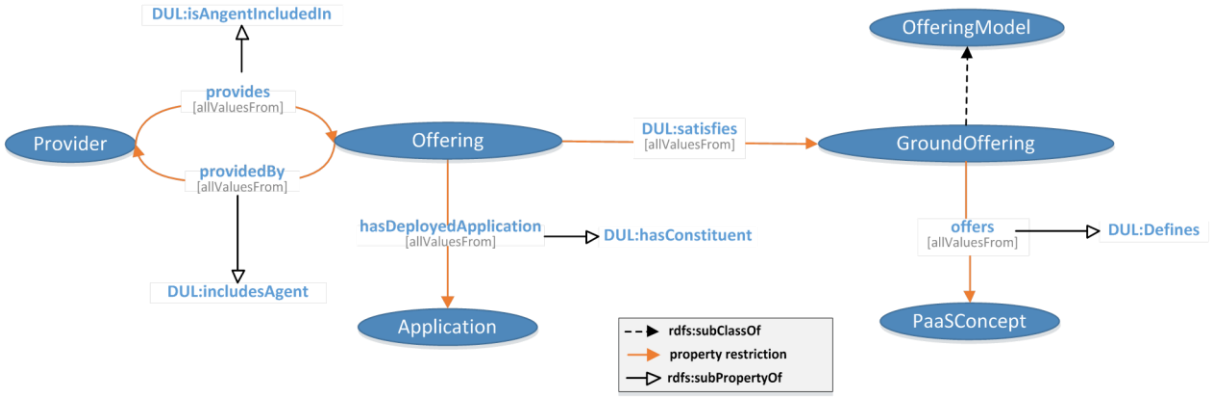

Figure 3. PaaSport Offering model.

The application model enables the semantic annotation of developers' applications. Developers can define functional (programming language, servers, database etc.) and non-functional (performance, capacity etc.) characteristics that the deployment environment should satisfy. These characteristics are used by the match-making algorithms to match PaaS Offerings that are the most relevant to their application requirements. The extension of the abstract PaaSport model is presented in **Figure 4**.

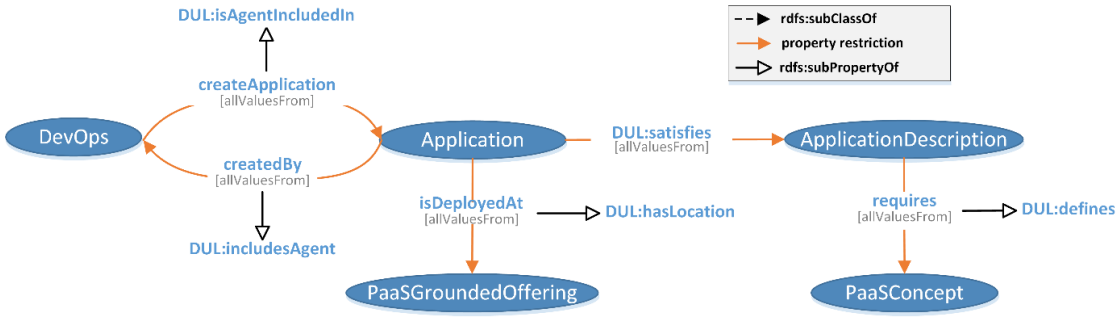

Figure 4. PaaSport Application model.

If we want to add a new parameter to a concept we have only to declare the quality value of this parameter. Moreover, we could do the same for concepts. The application example in Figure 5 describes the logic underlying the PaaSport semantic models. Specifically, we zoom into the defintion of a QoS parameter, namely the network latency, whose value is interpreted as a numerical range value with an upper value (Max) and it is measured in milliseconds.

One of the most important classes in the ontology is **PaaSConcept**, which is a basic unit of an offering (or application requirement) and represents an abstract concept of a service. For the application developer it represents a requirement for the application and for the PaaS provider it is a part of an offering. Each PaaSConcept is associated with one or more PaaS parameters, through the DUL:hasParameter property. Concepts can be backwards compatible with other concepts (through the isCompatibleWith property). This is mainly needed for concepts such as versions of offered programming languages or services/software.

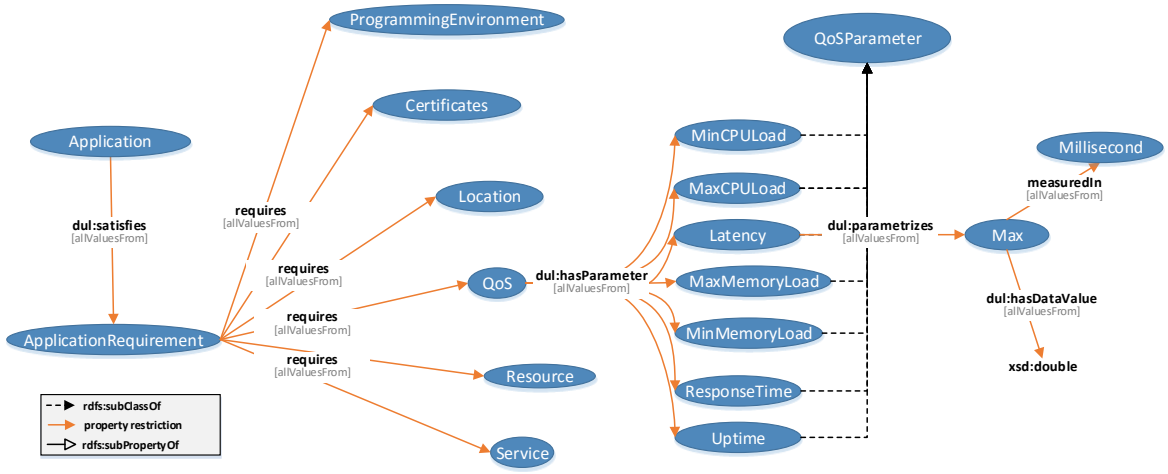

Figure 5. An Application example.

A parameter is a property of a concept. Specifically, a parameter classifies a concept, specifying the way it should be interpreted. For example, the value "0.09 seconds" refers to the Latency of the provided service. The value of the parameter is defined using the **hasParameterDataValue** property. Through OWL restrictions we associate parameters with specific PaaSConcepts. A **PaaSParameter** can be:

- a **MatchmakingParameter** that represents a parameter participating in matchmaking and ranking;
- an **InformationalParameter** used only for informational reasons and can be inspected manually by the application developer in order to select an offering.

Moreover, the matchmaking parameters are divided into functional and non-functional ones. This is achieved via the **FunctionalParameter** class: its subclasess are parameters that can only be used as functional requirements; otherwise, they can be used both as functional and non-functional ones. When functional requirements are not met by an offering, then it cannot be considered as a candidate for deploying an application. Non-functional parameters usually measure the quality of a service and are used to rank the order of preference of the selected services. Notice that a non-functional parameter can also be used as a functional one if the user wishes to. For example, if "latency less than 10ms" is absolutely required, offerings that do not satisfy this criterion are not considered at all. This can be declared by the user through the GUI.

The Measurement Unit Ontology[1] has been used for semantically representing the various measurements and units in PaaSport. The ontology can be used for modelling physical properties or qualities. Every unit is related to a particular kind of property. For instance, the Hz unit is uniquely related to the frequency property. Under the provided ontological approach, units are abstract spaces used as a reference metrics for quality spaces, such as physical qualia, and they are counted by some number. For instance, weight-units define some quality spaces for the weight-quality where specific weights of objects, like devices or persons, are located by means of comparisons with the proper weight-value of the selected weight-unit.

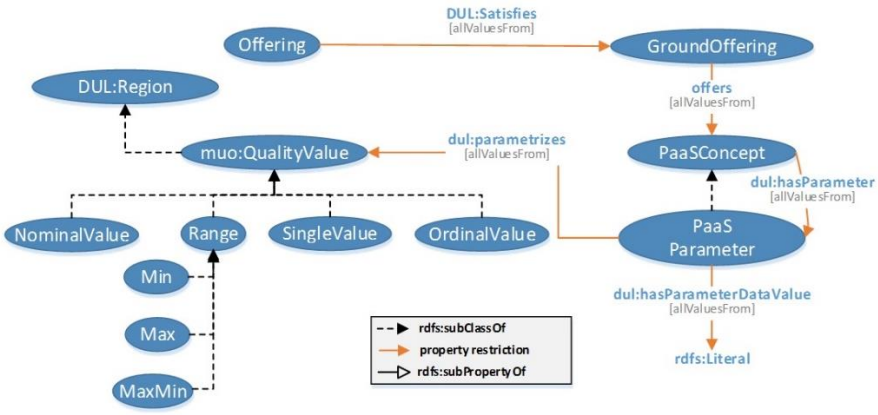

Figure 6. Hierarchy of quality values.

In MUO, the class muo:QualityValue is used for representing the values of qualities, for instance, the amount of available memory. Instances of this class are related with 1) exactly one unit, suitable for measuring the physical quality (meters for length, grams for weight, etc), by means of the property muo:measuredIn, 2) a number, which express the relationship between the value and the unit by means of the rdf:value property; 3) a time, which expresses the quality value along the line of time. In PaaSport, we use MUO to represent the units as well as qualitative attributes, whereas values are represented using the DUL vocabulary (dul:hasParameterDataValue). More specifically, there are the following quality values (**Figure 6**):

- Single Values, either symbolic or numeric, that require an exact match.
- Nominal Values, which are enumerated data types and require an exact match.
- Ordinal Values, namely ordered enumerated data types which also require exact match, but order can be established for better or worse.
- Range Values, which are numeric values that requires range match, e.g. “less than” or “equal”. There are 4 subclasses of this class, according to the matchmaking profile of each parameter.
    - Max: Range Value with a Max upper limit. Matches less than or equal.

[1] http://idi.fundacionctic.org/muo/

- Min: Range Value with a Min upper limit. Matches “greater than” or “equal”.
- MaxMin: Range Value with a limit that is Max for the Offering and Min for the Application. Matches “less than” or “equal”.
- MinMax: Range Value with a limit that is Min for the Offering and Max for the Application. Matches greater than or equal.

4 PaaSport Recommendation Algorithm

The Recommendation Layer of the PaaSport Reference Architecture (Section 3) involves the development of algorithms and software for supporting the selection of the most appropriate PaaS offering that best matches the requirements of the application a developer wants to deploy. Under this context, the PaaSport recommendation algorithms and models are aimed at providing the necessary semantic layer on top of the offering and application model descriptions, solving interoperability issues and improving the quality of the recommendations. To this end, standard vocabularies and ontology languages are used for capturing the structural and semantic characteristics of the various entities involved in the PaaSport domain, whereas the underlying conceptual models facilitate the use of lightweight reasoning during the matchmaking process.

This section describes the algorithms for PaaS Offering Matchmaking. More specifically, capitalizing on the Semantic Models presented in Section 3.1, we describe the capabilities of the matchmaking algorithms in terms of the supported semantics and filtering mechanisms that are used for selecting PaaSport Offerings. We also describe the implementation of the matchmaking procedure, elaborating on the SPARQL queries we have developed for querying the semantic repository, and we present matchmaking examples. The PaaSport Recommendation Algorithm has been developed in Java using the Apache Jena (Apache Jena, 2016) framework for parsing ontologies, processing data and executing SPARQL queries. Finally, we describe how the algorithm and the semantic models are interoperating with the persistence layer of the PaaSport marketplace.

Notice that the recommendation algorithm searches for any (single) platform offering of a (single) cloud provider, among multiple cloud providers, that offers all the compatible requested services. The restriction to a single platform offering / cloud provider is not inherent in our algorithm, but is imposed by the business model of the PaaSport Marketplace and the technical solution for interoperability / portability implemented by the PaaSport Broker. Usually PaaS providers charge for a platform as a whole and not as single provided services. Therefore, when a DevOps engineer wants to deploy / migrate his / her application from one cloud provider to another, usually searches for a better (more cost efficient) platform. This offers a cleaner deployment and maintenance solution. However, the recommendation algorithm can be easily extended to

multiple cloud providers at the service level. This extensibility is also indicated in various places in this section.

4.1 Matchmaking and Ranking Algorithm

At the heart of the PaaS Offering Recommendation Layer there is a recommendation algorithm that selects and scores-ranks the most appropriate PaaS offerings that best match the requirements of the application a developer wants to deploy. The matchmaking and ranking algorithm consists of two steps (**Figure 7**):

1. Selection of those offerings that satisfy the functional parameters.
2. Scoring of the remaining (from step 1) offerings using an aggregation scoring function on all the non-functional parameters.

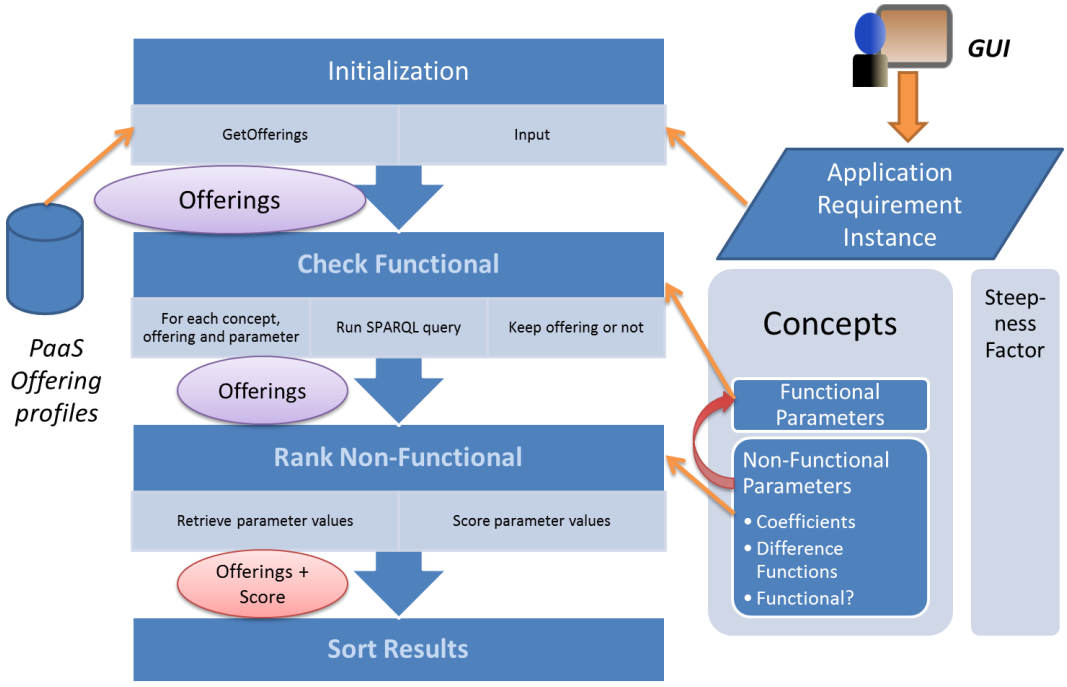

Figure 7. Overview of The PaaSport Matchmaking and Recommendation algorithm

Note that when talking about functional and non-functional parameters, we refer to the parameters that the DevOps Engineer has set as application requirements through the corresponding GUI. Also note that parameters are classified as either functional or non-functional from the PaaS Semantic Model (Section 3.1). So, the GUI can be constrained by the Model on which parameters can be used as functional or non-functional. However, non-functional parameters can be used both as non-functional and functional. For example, one might require that the storage capacity of the offering should be no less than 10GB and that he/she is not willing to consider offerings in the final ranked list with less storage, even with a lower score than the others. In this case, the parameter will be included twice in the list of parameters retrieved by the GUI, once

in the functional parameters list and once in the non-functional parameters list. However, the opposite is not allowed, i.e. a functional parameter (set by the Semantic Model, e.g. the Programming Language) can never be treated as non-functional.

The algorithm implemented in the PaaS Offering Recommendation Layer accesses the PaaS Offerings profiles stored in the Persistence and Execution Layer. Regardless of the implementation choices, the best way to do this is through SPARQL queries, which preserve the semantics of the conceptual RDF graph model. Thus, the algorithm was designed in such a way, that it uses predefined SPARQL templates in order to query the offering profiles, according to the parameter type.

Another design decision we followed is to use many "compact" (i.e. small) SPARQL queries instead of using a few "big ones". This decision has to do with the ignorance of the underlying persistence layer and the optimizations they provide for large SPARQL queries. Another reason for this decision is to be able to make fewer iterations and therefore queries, when there are offerings that violate one functional parameter early in the process. In order to have a better insight, imagine that the general logic behind checking the functional parameters is that "selectable" offerings are the ones that satisfy all functional parameters:

$$\forall Off, \forall Par \in Functional(App) \rightarrow satisfies(Off, Par)$$

If the above logical expression is naively turned into an algorithm, then it could be implemented as either:

1. $N*R$ SPARQL queries for checking satisfaction of one parameter for each offering (N is the number of offerings, R is the number of functional parameters);
2. N big intersection SPARQL queries for checking satisfaction of all R parameters for each offering;
3. one very big intersection SPARQL query for retrieving all offerings that satisfy all R functional parameters.

Intuitively, options 2 and 3 above would provide very complex queries that could be difficult to run efficiently on RDF triplestores, and even more difficult to run on relational database systems (in case the option of mapping the relational database schema to a virtual RDF graph is chosen for the semantic layer). However, option 1 also runs unnecessarily many SPARQL queries, whereas our algorithm tries to run fewer SPARQL queries by pruning early offerings that do not satisfy one functional parameter.

The developed algorithm, in summary, iterates over functional parameters and offerings and, whenever an offering violates a functional parameter, it is excluded from the rest of the iterations. This means that, if (as a best case) an offering violates the first functional parameter that is examined, then the rest R-1 parameters will not be checked; therefore, R-1 less SPARQL queries of type 1 will be executed. As an average case we may consider that half the offerings will be excluded midway (e.g. checking half the parameters); therefore $N*R/4$ less SPARQL queries will be executed. If we consider specific values N=1000 and R=10, then this

would result in 2500 fewer SPARQL queries. Another extreme case is when no offering exists that satisfies all the functional parameters. Using the naive approach, *N***R* queries would be needed, whereas with our approach (as a best case) only *N* queries suffice (suppose the first parameter is violated by all offerings).

4.1.1 Description of the Algorithm

Our algorithm, shown as function *getRecommendations* in **Algorithm 1**, takes as input the application requirements instance as defined by the DevOps Engineer from the GUI (?*GUI.AppInst*) and returns as output the list of offerings that are compatible with the functional requirements (?*results*), sorted in descending order according to the scoring function, which is based on the non-functional requirements.

The algorithm initially retrieves the IDs of all offerings stored in the Persistence Layer and stores them in-memory into the offerings list (?*offerings* - line 2). Next, we create a candidate list (?*candidates*) consisting of pairs <?*o*,?*r_type_list*> that represent all candidate offerings and all required concept types for each offering (lines 4-8). The first loop iterates every offering, initializes the concept list to null, and then iterates over all concepts ?r of the application requirement (*GUI.AppInst.satisfies.requires*), keeping their types (?*r.type*) in the ?*r_type_list*. Note that each member of ?*r_type_list* is also a pair <?*r.type*,?*concept-list*>; ?*concept-list* is a list of concept instances of the same type for the same offering. For example, imagine that there might be several database options offered by a PaaS offering. This, according to the PaaS Semantic Model (described in deliverable 1.3), means that the concept type *paas*:*Database* will have many instances attached to the offering instance; each Database concept instance stands for a different database offering (e.g. MySQL, PostgreSQL, etc.). Initially, the concept list is empty; it will be filled later with concept instances that do not violate the functional requirements.

Then, there are two main loops in the algorithm that implement the two steps mentioned above:

1. Matchmaking using Functional Parameters (lines 9-24), and
2. Ranking using Non-Functional Parameters (lines 31-62 and 63-79),

with some housekeeping lines in between. The following subsections describe both of these steps in detail.

Algorithm 1. The main algorithm for matchmaking and ranking offerings.

```
     Function getRecommendations(?GUI.AppInst) : ?results:ListOfOfferings
1.   ?K = 5
2.   ?offerings = getPaaSOfferings()
3.   ?candidates = ∅
4.   For each ?o in ?offerings
5.        ?r_type_list = ∅
6.        For each ?r in GUI.AppInst.satisfies.requires
7.             ?r_type_list = ?r_type_list ∪ <?r.type, ∅>
8.        ?candidates = ?candidates ∪ { <?o, ?r_type_list> }
9.   For each ?r in GUI.AppInst.satisfies.requires
10.       For each < ?o, ?r_type_list > in ?candidates
```

```
11.                 Find <?r.type, ?concept-list> ∈ ?r_type_list
12.                 For each ?p in ?r.hasParameter such that ?p ∈ GUI.FunctionalParameters
13.                     ?sparql-templ = retr-functional-sparql-templ(?p.parametrizes.type)
14.                     ?result = Run-functional-sparql-templ(?sparql-templ, ?o, ?r.type, ?p.type, ?p.Value, ?p.qualityValue,
15.                                 ?p.qualityValue.MeasureUnit)
16.                     ?candidates = ?candidates \ { <?o, ?r_type_list > }
17.                     If ?concept-list=∅
18.                         Then ?new-concept-list = ?result
19.                         Else ?new-concept-list = ?concept-list ∩ ?result
20.                     If ?new-concept-list == ∅
21.                         Then Break
22.                         Else ?r_type_list = ?r_type_list \ { <?r.type,?concept-list> } ∪ { <?r.type, ?new-concept-list };
23.                             ?candidates = ?candidates ∪ { <?o, ?r_type_list> }
24.         If ?candidates == ∅ Then Break
25.     If ?candidates == ∅ Then Return ∅
26.     ?results =?values = ?minmax = ∅
27.     For each ?p in ?r.hasParameter such that ?p ∈ GUI.NonFunctionalParameters
28.         ?min = unbounded
29.         ?max = ?prevmax = 0
30.         ?minmax = ?minmax ∪ <?p, ?min, ?prevmax, ?max>
31.     For each ?r in GUI.AppInst.satisfies.requires
32.         For each <?o,?r_type_list> in ?candidates
33.             Find <?r.type, ?concept-list> ∈ ?r_type_list
34.             ?sparql-templ = retr-non-functional-sparql-templ(?concept-list)
35.             For each ?p in ?r.hasParameter such that ?p ∈ GUI.NonFunctionalParameters
36.                 ?ValueSet = Run-nonfunctional-sparql-templ(?sparql-templ, ?o, ?concept-list, ?r.type, ?p.type,
37.                             ?p.qualityValue, ?p.qualityValue.MeasureUnit)
38.                 ?temp-Values = ∅
39.                 ?temp-Concept-Values = ∅
40.                 For each <?v,?c> in ?ValueSet
41.                     ?temp-Values = ?temp-Values ∪ {?v}
42.                     If <?c,?Vals> ∈ ?temp-Concept-Values
43.                         Then ?temp-Concept-Values = ?temp-Concept-Values \ <?c,?Vals> ∪ {<?c,?Vals∪{?v}>}
44.                         Else ?temp-Concept-Values = ?temp-Concept-Values ∪ {<?c, {?v}>}
45.                 ?Concept-Values = ∅
46.                 For each <?c,?c-vals> in ?temp-Concept-Values
47.                     ?v = Find_best(?c-vals, ?p.parametrizes.type, ?p.Value)
48.                     If <?c,?p-values> ∈ ?Concept-Values
49.                         Then ?Concept-Values = ?Concept-Values \ {<?c,?p-values> } ∪ {<?c, ?p-val-
50.                                 ues∪{<?p,?v>}> }
51.                         Else ?Concept-Values = ?Concept-Values ∪ {<?c, {<?p,?v>}> }
52.                 ?values = ?values ∪ <?o, ?Concept-Values >
53.                 ?v_min = Find_min(?temp-Values, ?p.parametrizes.type, ?p.Value)
54.                 ?v_max = Find_max(?temp-Values, ?p.parametrizes.type, ?p.Value)
55.                 Find <?p, ?min, ?prevmax, ?max> ∈ ?minmax
56.                 If (?v_max == unbounded OR (?max !== unbounded AND compare_greater(?v_max, ?max, ?p.para-
57.                             metrizes.type, ?p.Value)))
58.                     Then ?prevmax = ?max; ?max = ?v_max
59.                     Else If (?min == unbounded OR (?v_min !== unbounded AND compare_greater(?min, ?v_min,
60.                     ?p.parametrizes.type, ?p.Value)))
61.                         Then ?min = ?v_min
62.                 ?minmax = ?minmax \ { <?p, ?min, ?prevmax, ?max> } ∪ <?p, ?min, ?prevmax, ?max>
63.     For each <?o,?r_type_list>  in ?candidates
64.         Find <?o, ?Concept-Values > ∈ ?values
```

```
65.          For each ?r in GUI.AppInst.satisfies.requires
66.               Find <?r.type, ?concept-list> ∈ ?r_type_list
67.               ?MaxConceptScore = 0
68.               For each <?c, ?p-values> ∈ ?Concept-Values such that ?c ∈ ?concept-list
69.                    ?conceptScore = 0
70.                    For each <?p,?v> ∈ ?p-values
71.                         Find <?p, ?min, ?prevmax, ?max> ∈ ?minmax
72.                              ?score = Score-offering(?v, ?p.Value, ?p.parametrizes.type, ?min, ?prevmax, ?max,
73.                                                                   ?p.coefficient, ?p.differenceFunction, ?K)
74.                         ?conceptScore = ?conceptScore + ?score
75.                    If ?conceptScore > ?MaxConceptScore
76.                         Then ?MaxConceptScore = ?conceptScore
77.               If <?o, ?oldScore> ∈ ?results
78.                    Then ?results = ?results \ <?o, ?oldScore> ∪ <?o,?oldScore+?MaxConceptScore>
79.                    Else ?results = ?results ∪ <?o, ?MaxConceptScore>
80.     Sort ?results on Descending ?score
81.     Return ( ?results )
```

Notice that the above algorithm can be easily tweaked to return (the best) services of multiple cloud providers, instead of returning an offering of a single cloud provider, as a whole. This can be achieved by eliminating all lines that have to do with offerings, such as lines 2-8, the second loop of the matchmaking in line 10, the second loop in the scoring gathering part (line 32), the first loop in scoring calculation part (line 63), and so on, so forth. Of course, the SPARQL templates described in the remaining of this section should also be tweaked, as explained later.

Matchmaking using Functional Parameters

The first step of the algorithm, where offerings are filtered according to whether they satisfy the functional parameters or not (lines 9-24), consists of three nested loops. The first loop iterates over all PaaS concepts ?*r* of the Application Requirement (retrieved by the GUI) (line 9), whereas the second loop iterates over all offering entries in the candidate offerings set (or list) (line 10). Note, that since offerings might be removed later on, inside the third loop, the candidate offerings set may be smaller at each iteration of the first loop.

The third loop (line 12) iterates over all functional parameters ?*p* of the currently examined application's concept ?*r*. We retrieve the appropriate SPARQL template for checking a functional parameter, according to the type of the Quality Value (Single, Range, Nominal, or Ordinal) that parameterizes the property (?*p.parametrizes.type*) and then we execute the retrieved SPARQL template, grounding placeholders with values from the corresponding parameter of the application requirement (lines 13-15). More details about the SPARQL templates are given later in section 4.2.

The result of the SPARQL query is a set of concept instances of the offering that satisfy the functional parameter. The concept list of the current offering ?*o* and the current concept type ?*r.type* is updated (lines 16-23), considering that the offering might violate this functional parameter or that the conjunction of the

functional parameters for the current concept type is not satisfied by any concept instance of the offering. First, the previous entry for the offering is deleted from the candidate list (line 16) and it will be re-added if the offering is still valid. Note that initially, when the concept list is empty, the updated concept list is set to the result returned by the query (lines 17-18). Otherwise, the updated concept list is constructed as the intersection of the previous list and the new set of results, meaning that the remaining concept instances should satisfy all functional parameters of the concept type (line 19). Intersection though may produce empty sets, either because ?*result* is empty or because the concept instances already existing in the list do not coincide with the concept instances that satisfy this functional parameter, i.e. there are no concept instances that conjunctively satisfy all the functional parameters of the current concept type. In either case, the offering is not re-added to the candidate offerings list and it will not be checked any more in future iterations. In this case there is also an exit from the current (third) loop, because there is no need to check more functional parameters for this offering (lines 20-21). If the updated concept instance list is not empty, then the offering is added back to the candidate list, along with the updated concept list (lines 22-23).

Upon the termination of the middle (second) loop, there might be the case that there are no more candidates left. Then the outermost loop is exited prematurely (line 24). When the outer loop terminates, we should have all offerings/concepts that satisfy all the functional parameters of the application profile. In case there are no more candidates left, then the *getRecommendations* function just returns an empty list and does not continue with ranking (line 25).

Ranking using Non-Functional Parameters

The second step of the algorithm, which scores and ranks the offerings that satisfy the functional parameters, using the non-functional parameters, consists of four sub-steps:

1. Initialization of the auxiliary data structures used in the main algorithm (lines 26-30).
2. Retrieval of the values of non-functional parameters in the auxiliary data structures (lines 31-62).
3. Calculation of the score of each offering using the auxiliary data structures (lines 63-79).
4. Sorting of the offerings in descending order of their score (lines 80-81).

Initialization of auxiliary data structures

The auxiliary data structures are the following:

- The variable ?*results* is a set of <?*o*, ?*Score*> pairs, where ?*o* is an offering and ?*Score* is its score.
- The variable ?*values* is a set of <?*o*, ?*Concept-Values* > pairs, where ?*o* is an offering and ?*Concept-Values* is a set of concept instances and the values of non-functional parameters that each instance contains. So, ?*Concept-Values* consists of pairs <?*c*,?*p-values*>, where ?*c* is a concept instance of ?*o* and ,?*p-values* is a set of parameter values, in the form of <?*p*,?*v*>, where ?*p* is a non-functional

parameter of ?*c* and ?*v* is its value. Notice that we keep only one value for each parameter in this structure. In case there is more than one value, we keep the "best" (explained later).

- The variable ?*minmax* is a set of tuples of the form <?*p*, ?*min*, ?*prevmax*, ?*max*>, where ?*p* is a non-functional parameter, ?*min* / ?*max* are the minimum / maximum values for this parameter, along with all the offerings that satisfy the functional parameters, and ?*prevmax* is the second to maximum value, which is needed in the scoring function in case the maximum value is "unbounded".

During the initialization sub-step (lines 26-30), ?*results* and ?*values* are set to empty, while ?*minmax* is initialized with tuples for all non-functional parameters. The initial values are "unbounded" for ?*min* and 0 for ?*max* and ?*prevmax*.

Retrieval of non-functional parameter values

The second sub-step of the ranking step (lines 31-62), consists of three main nested loops, and two smaller consecutive loops, nested inside the third loop. The first loop iterates over all PaaS concepts ?*r* of the Application Requirement (line 31), while the second loop iterates over all offerings that "satisfy" the functional requirements (line 32). For each offering we retrieve the appropriate set of concept instances (?*concept_list*) that corresponds to the same concept type (?*r.type*) with the application profile (line 33). We retrieve the appropriate SPARQL template for retrieving a non-functional parameter (line 34). There is only one template (see section 3.4.2); however, if the concept list is empty, then this indicates that concept ?*r* of the application profile has only non-functional parameters attached to it and no functional parameters, therefore the VALUES construct will not appear at all in the SPARQL template (explained also in section 4.2.2).

The third loop (line 35) iterates over all non-functional parameters of the concept ?*r*. We run the SPARQL template in order to retrieve the values of each parameter of the offering (lines 36-37), along with the concept instance they belong to. Placeholders are grounded with values from the corresponding parameter of the application requirement. Note that, since an offering can contain alternative options for the same concept (e.g. several storage options, based on different pricing policies), or alternative concept instances for the same concept type, the returned result could be a set (?*ValueSet*) of value-concept-instance (<?*v*,?*c*>) pairs of the same concept type. Also, note that the concept list of compatible concepts (of the same concept type) with the functional parameters is given, in order to avoid retrieving parameter values of non-functional parameters of concepts that violate the functional parameters. The concept list can be empty in cases when the requirement contains only non-functional parameters. In this case, the ?*concept-list* parameter is ignored (the SPARQL template retrieved previously does not contain the VALUES construct).

We construct two new temporary auxiliary data structures (?*temp-Values*, ?*temp-Concept-Values*, lines 38-39) in order to find the minimum / maximum values for each parameter and in order to construct the <?*p*,?*v*> pairs for each offering and concept instance (see above), respectively. For each <?*v*,?*c*> element of the result

we store the values in ?*temp-Values* and we store (and update) a <?*c*,?*Vals*> pair for each concept instance ?*c* in ?*temp-Concept-Values* (lines 40-44). If many alternative values (?*c-vals*) for the same parameter (?*p*) and for the same concept instance (?*c*) are offered by the offering, we only keep the "best" (?*v*) of them (for scoring purposes), in lines 46-51. The "best" is defined according to the type of the quality value that parametrizes the non-functional parameter (line 47). The definition for the *find_best* function is displayed in **Algorithm 3**. Eventually, the ?*Concept-Values* set (discussed above) is constructed for all non-functional parameters of every concept that "satisfies" the functional requirements (lines 48-51) and the ?*values* set is updated (line 52). Lines 53 and 54 calculate the minimum and maximum value in ?*temp-Values*, according to the type of the quality value that parameterizes the non-functional parameter; these functions are defined similarly to *find_best* (**Algorithm 3**). Lines 55-62 update this maximum/minimum values. Things are complicated because a) the comparison is performed according to the type of the quality value that parameterizes the non-functional parameter (lines 56 and 59-60), using **Algorithm 5**, and b) even for numerical values we have to cater for the case when the value is "unbounded" (lines 56 and 59).

Score and Sort all offerings

The third sub-step is to calculate the score of each offering using the auxiliary data structures (lines 63-79). There are 4 nested loops. The outer loop iterates over all offerings in the candidate list (line 63). For each offering (?o) its ?*Concept-Values* set is retrieved (line 64). The second loop iterates over all PaaS concepts (?*r*) of the Application Requirement (line 65) and retrieves the appropriate concept list (line 66) for the current concept type (?*r.type*). The third loop (line 68) iterates over all concept instances (?*c*) of the offering found in ?*Concept-Values* that match the ones found in the concept list of the appropriate concept type. The fourth loop (line 70) iterates over all property-value (<?*p*,?*v*>) pairs for each concept instance (?*c*) and calculates the score for each parameter of the concept, using the scoring function (lines 72-74) analyzed in Section 4.1.2. The features that play role in scoring are: a) the value of the parameter, b) the minimum/maximum parameter values, c) the parameter value of the application requirement, d) the coefficient of the parameter as defined by the user (through the GUI), e) the difference function as defined by the user (through the GUI), which defines how important the differences are between the value of the offering and the value of the requirement (linear, sublinear, superlinear, etc), f) the type of the quality value that parametrizes the non-functional parameter (e.g. max or min) and, finally, g) how steep the sub- and super- linear curves are (factor ?*K*). Note that in line 74 we calculate the score per concept instance, and then, outside the fourth loop but inside the third (lines 75-76) we keep track of the concept instance with the maximum score (per concept type), which is used to calculate the offering score outside the third loop (but inside the second) in lines 77-79. Finally, when the above procedure ends, all offerings have been scored. Line 80 sorts the offerings in descending order according to the score and line 81 returns the results.

4.1.2 Scoring Function

The scoring function for each non-functional parameter actually tries to measure how far the value of the current offering is from the worst offering (for this parameter), so that offerings can be compared in a fair manner according to this parameter. Furthermore, in order for the comparison to be fair across all parameters, this function always returns a value between 0 and 1. Of course, since each parameter counts differently according to the DevOps engineer, the value of this scoring function is multiplied by a weighing factor. The sum of all weights for all parameters equals to 1.

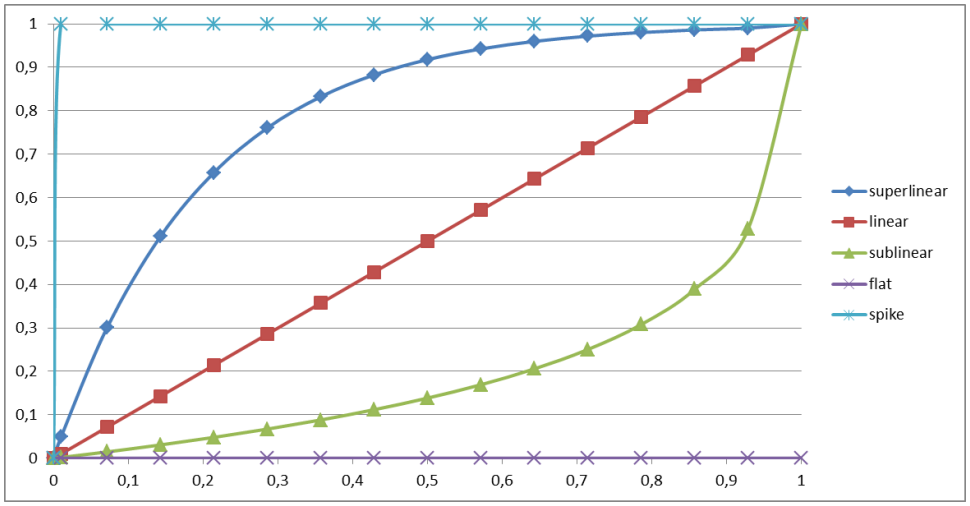

Figure 8. The difference functions supported by scoring function.

The scoring function for each non-functional parameter takes as input:

- the Minimum (?*min*) / Maximum (?*max*) / Previous of Maximum (?*prevmax*) values for the offerings parameters that satisfy the functional requirements,
- the type (?*parametrizes_type*) of the Quality Value (Single, Nominal, Ordinal, Range) that parametrizes the non-functional parameter,
- the coefficient (?*coefficient*) of the parameter as defined by the user (through the GUI),
- the name of the difference function (?*differenceFunction*) as defined by the user (through the GUI), which defines how important the differences are between the value of the parameter, the current offering and the corresponding value of the worst offering (linear, sublinear, superlinear, etc.), and
- a configuration parameter (?*K*) that defines how steep the sublinear and superlinear curves are, which could be adjusted by the admin or even the user from the GUI.

The score of an offering for a parameter is given by the following function:

$$ParScore_{par}(OffParVal) = \begin{cases} 1, & OffParVal = \infty \\ 0, & OffParVal < AppReqParVal \\ w_{par} f_{df_{par}}(x_{OffParVal}), & otherwise \end{cases}$$ **Eq.** 1

where, *OffParVal* is the value of the parameter for the offering, *AppReqParVal* is the required value for the parameter by the application, w_{par} is the coefficient of the parameter, and f_{dfpar} is the difference function selected for the parameter. In case the value of the parameter for the offering is unbounded (∞) it means that it should be assigned the maximum scoring value (1). On the other hand, if the value of the offering parameter is less than what the application requires, then it is assigned the minimum scoring value (0). In any other case, the score of the difference function is multiplied by the corresponding weight.

We support the following difference functions, which are better illustrated in **Figure 8**:

- *Flat Scoring Function*: $f_{fl}(x) = 0$, meaning that no matter what the value of the input is, the output is always zero. This behavior could be equally well produced by zero weight or just by eliminating the parameter from the GUI. We provide it for the sake of completeness.
- *Linear Scoring Function*: $f_{lin}(x) = x$. The most usual case; the output is analogous to the input, meaning that doubling the input, the output also doubles.
- *Super-linear Scoring Function*: $f_{sup}(x) = 1 - e^{-kx}$. In this case, the output initially grows exponentially with the growth of the input. This means that small differences in this parameter should be scored highly. An example of this could be for scarce and expensive resources, as e.g. memory capacity. A double memory capacity could mean 10 times better offering (according to this criterion alone). The evolution of this super-linear function is controlled by a constant *k*, which could be selected by the PaaSport administrator or even the DevOps engineer. In the latter case, it should be the same for all parameters, for the sake of fairness. In **Figure 8** the factor *k* was set to 5. Note, that since *f*(*x*) can never exceed 1, this function rises quickly and then saturates to 1, meaning that beyond a certain point there is no added value to increase this parameter.
- *Sub-linear Scoring Function*: $f_{sub}(x) = f_{sup}^{-1}(x) = -\frac{\ln(1-x)}{k}$. This case is symmetrical to the super-linear function, using the linear function as the symmetry axis. This literary means that this function is the inverse of the super-linear one. The output initially grows very slowly (logarithmically) with the growth of the input. This means that big differences in this parameter should not be assigned a high scoring value. An example of this could be for abundant and cheap resources, as e.g. HDD storage capacity. Ten times the storage could mean just 2 times better offering (according to this criterion alone). The evolution of this sub-linear function is also controlled by the constant k. Note, that since this function is symmetrical to the super-linear one, it rises quickly near x=1 to reach f(1)=1 at the end.

- *Spike Scoring Function*: $f_{sp}(x) = \begin{cases} 0, x = 0 \\ 1, x > 0 \end{cases}$. In this case, when the input is minimum, so is the output (0), whereas at all other cases the output is maximum (1). Actually it is the step function that indicates that whatever better than the minimum is scored highly.

The input value *x* of the difference function is related, of course, to the value of the non-functional parameter of the offering. It also depends on the type of the quality value that parametrizes the nonfunctional property.

$$x = \begin{cases} a, & parameter.type = Min \vee MaxMin \\ 1 - a, & parameter.type = Max \vee MinMax \end{cases}$$ **Eq.** 2

The first case is when the greater the value of the offering, the better the score should be, as in e.g. storage capacity. The second case corresponds to the opposite situation: the lesser the value of the offering, the better the score should be, as in e.g. network latency.

Quantity *a* is always in the range of 0-1 (as it is also the case for x). In order to be so, it must be defined via a ratio whose denominator is greater or equal to the nominator.

$$a = \begin{cases} \frac{(OffParVal - MinOffParVal)\left(1 - \frac{1}{MaxDiff}\right)}{MaxDiff}, & MaxOffParVal = \infty \\ \frac{OffParVal - MinOffParVal}{MaxDiff}, & otherwise \end{cases}$$ **Eq.** 3

The second case is the typical one, where the difference between the parameter value of the offering and the minimum parameter value of all offerings is divided by the maximum difference of this specific parameter domain (only the offerings that satisfy the functional parameters are members of the domain). The first case is a special case where the maximum value of the domain is unbounded (∞). In this case, we use the second greatest value of the domain; however, we also subtract a small quantity (that depends on the values of the domain) from the difference in the nominator, so that score 1 can only be assigned to the actual maximum value for the domain, i.e. the value unbounded. *MaxDiff* is calculated (for these two cases) as shown below:

$$MaxDiff = \begin{cases} PrevMaxOffParVal - MinOffParVal, & MaxOffParVal = \infty \\ MaxOffParVal - MinOffParVal, & otherwise \end{cases}$$ **Eq.** 4

Moving to the description of the actual algorithm for the scoring function, illustrated in **Algorithm 2**, we first find the "appropriate" operator for comparison (line 1), according to the type of the Quality Value that parametrizes the non-functional parameter. Then we calculate the appropriate ?*maxdix* (lines 2-4) according to a) if the ?*max* value is unbounded, then so is ?*maxdif*, or b) using the appropriate operator, as shown in **Algorithm 7**. Similarly, the ?*prevmaxdif* (see first case of Eq. 4, above) is calculated in line 5.

Algorithm 2. The scoring function for each parameter.

```
     Function Score-offering(?offering-value, ?app-value, ?parametrizes_type, ?min, ?prevmax, ?max, ?coefficient, ?dif-
     ferenceFunction, ?K) : ?Score:float
1.   ?Op = find-operator(?parametrizes_type)
2.   If ?max == unbounded Then ?maxdif = unbounded Else ?maxdif = difference(?op, ?max, ?min)
3.   ?prevmaxdif = difference(?op, ?prevmax, ?min)
4.   If ?offering-value == unbounded
5.         Then ?y = 1
6.         Else If !check-op(?op, ?offering-value, ?app-value)
7.               Then ?y = 0
8.               Else If ?op == "=="
9.                     Then ?y = 1
10.                    Else
11.                          If ?maxdif == unbounded
12.                                Then ?a = abs( difference(?op, ?offering-value, ?min)) *
13.                                                (1-1/?prevmaxdif) / ?prevmaxdif
14.                                Else ?a = abs( difference(?op, ?offering-value, ?min)) / ?maxdif
15.                          If ?Op == "<=" Then ?x = 1 – ?a
16.                          Switch ?differenceFunction
17.                                Case flat:          ?y = 0
18.                                Case sublinear:     If ?x == 1 Then ?y = 1 Else ?y = -ln(1 - ?x)/?K
19.                                Case linear:        ?y = ?x
20.                                Case superlinear:   ?y = 1-exp(-?K * ?x)
21.                                Case spike:         If ?x == 0 Then ?y = 0 Else ?y = 1
22.  ?Score = ?coefficient * ?y
23.  Return (?Score)
```

Then, if the offering features an unbounded value for the parameter (line 6), then the difference function scores 1 (the maximum – see Eq. 1, first case). Else, if the offering value is no better than the application requirement (line 9), which is determined using **Algorithm 6**, then the difference function scores 0 (the minimum – see Eq. 1, second case). If it is "better" (or equal), then check if the comparison is "pure" equality; if yes, then the difference function scores 1 (the maximum), else calculate quantity ?*a* (lines 13-16), according to Eq. 3, and then the actual input ?*x* to the difference function (lines 17-18), according to Eq. 2. Lines 19-31 implement the 5 different difference functions, presented above. The only thing to note is the case for the sub-linear function, which is not defined (log0) when ?x equals 1. In this case the function value is set to 1 (lines 23-24). Finally, the Score is calculated by multiplying the result of the difference function by the parameter's coefficient (user's weight factor) (line 32) and the score is returned.

4.1.3 Auxiliary Functions

In this section we present the algorithms for the most important auxiliary functions that are used in the recommendation algorithm. A full description of all auxiliary algorithms can be found at (PaaSport Consortium, 2014d). **Algorithm 3** defines the best of a set of values for a parameter according to the type of the quality value. It takes as input the set of values, the type of the parameter and the corresponding application

value. For numerical ranges (Min, MaxMin, Max, MaxMin) the "best" value is either the maximum numerical value (Min, MaxMin) or the minimum numerical value (Max, MaxMin), which can be found using trivial algorithms (not shown here for brevity). In case of ordinal values, a special function, shown in **Algorithm 4**, is used to find the maximum ordinal value. Note here that in the PaaSport semantic model, ordinal values are sorted in ascending order. Finally, Single and Nominal Values are treated alike: the "best" value is the value that equals the application value. If such a value does not exist in the set, then an arbitrary one is assigned.

Algorithm 3. Function that defines the best value according to the type of the quality value.

```
     Find_best(?ValueSet, ?parametrizes_type, ?app-value) : ?Value : anyType
1.   Switch ?parametrizes_type
2.       Case Min or MaxMin:   ?Value = Find_numerical_max(?ValueSet)
3.       Case Max or MinMax:   ?Value = Find_numerical_min(?ValueSet)
4.       Case Ordinal:         ?Value = Find_ord_max(?ValueSet)
5.       Otherwise:            ?Value = ?ValueSet[1]
6.                             ?n = 2
7.                             ?last = |?ValueSet|
8.                             While (?Value !== ?app-value and ?n <= ?last) do
9.                                 ?Value = ?ValueSet[?n]
10.                                ?n = ?n + 1
11.  Return (?Value)
```

Algorithm 4 retrieves the maximum of ordinal values. Actually, it does not differ from a trivial maximum-finding algorithm for numerical values. The only difference is the comparison operator in line 5, which is implemented by function *check_op*, shown in **Algorithm 6**. The minimum is defined similarly.

Algorithm 4. Function that finds the maximum value of ordinal values.

```
     Find_ord_max(?ValueSet) : ?MaxValue:anyType
1.   ?MaxValue = ?ValueSet[1]
2.   ?n = 2
3.   ?last = |?ValueSet|
4.   While (?n <= ?last) do
5.       If check_op(ord_max,?ValueSet[?n],?MaxValue) Then ?MaxValue = ?ValueSet[?n]
6.       ?n = ?n + 1
7.   Return (?MaxValue)
```

Algorithm 5 compares two values according to the type of the quality value. It takes as input the two values, the type of the parameter and the application value. For numerical ranges (Min, MaxMin, Max, MaxMin) the comparison if performed using the numerical "greater-than" operator. In case of ordinal values, the comparison operator in line 6 is implemented by function *check_op*, shown in **Algorithm 6**. Finally, Single

and Nominal Values are treated alike: if the first value equals the application value while the second value does not, then the first value is greater than the second. In all other cases, the comparison is false.

Algorithm 5. Function that compares two values according to the type of the quality value.

```
     compare_greater(?v1, ?v2, ?parametrizes_type, ?app_value) : ?check:Boolean
1.   ?check = false
2.   Switch ?parametrizes_type
3.        Case Min or MaxMin or Max or MinMax:   If ?v1 > ?v2 Then ?check = true
4.        Case Ordinal:                          If check_op(ord_max,?v1,?v2) Then ?check = true
5.        Otherwise:                             If ( ?v1 == ?app_value AND ?v2 !== ?app_value) Then ?check = true
6.   Return (?check)
```

Algorithm 6 checks whether the comparison between two values, using a certain comparison operator, is true. It takes as input the two values and the comparison operator. If the operator is a usual numerical comparison operator, then values are compared using this operator. If the operator is *ord_max*, meaning that the two values are ordinal and it should be checked if the first is greater than or equal to the second, then the SPARQL query of **Algorithm 15** is executed. In essence, this query checks if the first value follows the second in the directed graph of ordinal values. If the operator is the "equals" operator, the check is true if the two values are equal.

Algorithm 6. Check if comparison between two values using a comparison operator is true.

```
     check-op(?op, ?offering-value, ?app-value) : ?check:Boolean
1.   ?check = false
2.   Switch ?op
3.        Case ">=":        If ?offering-value >= ?app-value Then ?check = true
4.        Case "<=":        If ?offering-value <= ?app-value Then ?check = true
5.        Case ord_max:     If ( ?offering-value == ?app-value OR Run-check-ordinal-values-sparql(?offering-value,
6.                                    ?app-value) ) Then ?check = true
7.        Otherwise:        If ?offering-value == ?app-value Then ?check = true
8.   Return (?check)
```

Algorithm 7 calculates the difference between two values based on the comparison operator. It takes as input the two values and the comparison operator. If the operator is a usual numerical comparison operator, the difference is calculated using subtraction. If the operator is *ord_max*, meaning that the two values are ordinal, then if the two values are equal the difference is 0, otherwise we execute the SPARQL query of **Algorithm 16** that counts how many edges exist in the graph between the two ordinal values. If the operator is the "equals" operator, the difference is 0 if the two values are equal; otherwise difference is 1.

Algorithm 7. Calculate the difference between two values based on the comparison operator.

```
     difference(?op, ?v1, ?v2) : ?diff:numerical
1.   Switch ?op
2.       Case ">=" or "<=":     ?diff is ?v1 – v2
3.       Case ord_max:          If ?v1 == ?v2 Then ?diff = 0 Else ?diff = run-calc-ordinal-difference-sparql(?v1, ?v2)
4.       Otherwise:             If ?v1 == ?v2 Then ?diff = 0 Else ?diff = 1
5.   Return (?diff)
```

4.2 SPARQL templates

In this section we present the SPARQL templates that either check the functional parameters or retrieve the non-functional parameters. Finally, there is also a subsection that presents SPARQL queries needed for Ordinal values. Note that we suppose that there exists a function that returns the corresponding template, given the type of the quality value that parameterizes the parameters. Since this function is trivial to build, we do not present it here. Furthermore note that these templates have several placeholders (as e.g. <*offering.ID*>, <*concept.type*>, etc.), which are instantiated by the calling function before the SPARQL template is executed. Again, the function is trivial to build, so we do not present it here.

4.2.1 Checking Functional Parameters

All templates have a similar structure, returning the concept instances (?*concept*) that satisfy the functional parameter. If the offering does not satisfy the functional parameter, then such a query will return a NULL result set. This will be recognized by the matchmaking algorithm. The concepts that pass the functional parameters are registered by the calling algorithm.

No Quality Value

This is the simplest case, where no quality value exists for all. This could be for cases where a single value with no units is needed. Of course, the best way to deal with this would be through nominal values, but nevertheless we cover this case also for completeness (**Algorithm 8**). From the initial offering (<*offering.ID*>) the query navigates to its concepts (?*concept*) that share the same concept type (<*concept.type*>) with the application profile (lines 2-4). Then we retrieve all the parameters (?*par*) of the concept of the concept instance that share the same parameter type (<*par.type*>) with the application profile (lines 5-13). Note that we do not restrict the search only to parameters of the current concept instances, but also to parameters of compatible concepts, through the *paas*:*hasCompatibilityWith* transitive property (line 10). This could be used for cases e.g. of different programming language versions that are backwards compatible. Finally, in line 14 the value of the parameter is retrieved and in line 15 it is checked for equality against the corresponding value of application profile (<*par.value*>).

Algorithm 8. SPARQL template for checking a functional parameter without a quality value

```
1.  SELECT ?concept WHERE {
2.          <offering.ID> DUL:satisfies ?groundDescription .
3.          ?groundDescription paas:offers ?concept .
4.          ?concept rdf:type <concept.type> .
5.          {       ?concept DUL:hasParameter ?par .
6.          }
7.          UNION
8.          {       ?concept paas:hasCompatibilityWith+ ?concept1 .
9.                  ?concept1 DUL:hasParameter ?par .
10.         }
11.         ?par rdf:type <par.type> .
12.         ?par DUL:hasParameterDataValue ?Value .
13.         FILTER(?Value = <par.value> )
14. }
```

Notice that the above (and all consequent) SPARQL queries return offering concepts and parameters that have both exact and plugin conceptual relationships with the corresponding application concepts and parameters. This is due to the fact that triple patterns like "*?concept rdf:type <concept.type>* ." and "*?par rdf:type <par.type>* ." are interpreted under OWL semantics. However, there is no difference in scoring regarding either exact or plugin matching of concepts / parameters; plugin matching is considered as good as exact matching. Furthermore, the above (and all consequent) SPARQL queries can be easily tweaked to return services of multiple cloud providers by just eliminating lines 2 and 3, allowing thus for any concept (service) to be matched regardless of the offering that it belongs too.

Nominal Value

The nominal value case is similar to the previous one; the only difference being the addition of lines 13-14 (**Algorithm 9**) that checks if the quality value of the parameter of the application profile is indeed a nominal value and that it coincides with the quality value that parametrizes the parameter of the offering.

Algorithm 9. SPARQL template for checking a functional parameter with a nominal value

```
12.         ?par DUL:hasParameterDataValue ?Value .
13.         <par.qualityValue> rdf:type paas:NominalValue .
14.         ?par DUL:parametrizes <par.qualityValue> .
15.         FILTER(?Value = <par.value> )
```

Ordinal Value

The case of ordinal values differs from nominal values from line 14 onwards (**Algorithm 10**), since the compliance of an ordinal value parameter is that the offering has the same or better value than the one requested by the application. Lines 14-16 implement the check for the same value (similarly to nominal values), whereas lines 18-26 implement the "better value" case. For ordinal values, "better" means that the

offering value is an ontology instance that is either backward connected to the application request instance through the *DUL*:*precedes* property or that the application request instance is forward connected with the offering value instance through the *DUL*:*follows* property.

Algorithm 10. SPARQL template for checking a functional parameter with an ordinal value

```
14.     {       ?par DUL:parametrizes <par.qualityValue> .
15.             FILTER (?Value = <par.value> )
16.     }
17.     UNION
18.     {       ?par DUL:parametrizes ?QualityValue .
19.             ?QualityValue rdf:type paas:OrdinalValue .
20.             <par.qualityValue> DUL:precedes+ ?QualityValue .
21.     }
22.     UNION
23.     {       ?par DUL:parametrizes ?QualityValue .
24.             ?QualityValue rdf:type paas:OrdinalValue .
25.             ?QualityValue DUL:follows+ <par.qualityValue> .
26.     }
27. }
```

Single Value

Single values are arithmetic values that may be characterized by measurement units (e.g. 1 GB of memory). So, when comparing two values for equality, this should also include equality of units. But simply comparing both values and units is still not enough because two values can be different literally but equal conceptually if their units are convertible to each other. For example, 1024MB is exactly the same as 1GB, although neither the value nor the measurement unit is identical on the first sight. Hence the complexity of this SPARQL query (**Algorithm 11**). There are several alternative cases for equality; each one of them is one of the 5 graph patterns that are connected through the UNION operator.

The first case (lines 14-17) is where the quality value of the offering parameter is identical to the one of the application profile. This means that both values have identical units. Similar is the second case (lines 19-23) where the quality values are not identical but they are both measured with the same measurement unit (<*par.qualityValue.MeasureUnit*>). In both these cases, the values of the offering and the application will be compared directly, without conversions; this is reflected by the fact that variables ?*Factor1* and ?*Factor2* are both set to 1.

Algorithm 11. SPARQL template for checking a functional parameter with a single numerical value

```
14.     {       ?par DUL:parametrizes <par.qualityValue> .
15.             BIND (1 AS ?Factor1)
16.             BIND (1 AS ?Factor2)
17.     }
18.     UNION
19.     {       ?par DUL:parametrizes ?qualityValue .
20.             ?qualityValue uomvocab:measuredIn <par.qualityValue.MeasureUnit> .
21.             BIND (1 AS ?Factor1)
22.             BIND (1 AS ?Factor2)
23.     }
24.     UNION
25.     {       ?par DUL:parametrizes ?qualityValue .
26.             ?qualityValue uomvocab:measuredIn ?Units .
27.             <par.qualityValue.MeasureUnit> rdf:type ?AppParMeasureUnitType .
28.             ?Units rdf:type ?AppParMeasureUnitType.
29.             <par.qualityValue.MeasureUnit> rdf:type uomvocab:BaseUnit .
30.             ?Units rdf:type uomvocab:SimpleDerivedUnit .
31.             ?Units uomvocab:derivesFrom <par.qualityValue.MeasureUnit> .
32.             ?Units uomvocab:modifierPrefix ?prefix2 .
33.             ?prefix2 uomvocab:factor ?Factor2 .
34.             BIND (1 AS ?Factor1)
35.     }
36.     UNION
37.     {       ?par DUL:parametrizes ?qualityValue .
38.             ?qualityValue uomvocab:measuredIn ?Units .
39.             <par.qualityValue.MeasureUnit> rdf:type ?AppParMeasureUnitType .
40.             ?Units rdf:type ?AppParMeasureUnitType.
41.             <par.qualityValue.MeasureUnit> rdf:type uomvocab:SimpleDerivedUnit .
42.             ?Units rdf:type uomvocab:BaseUnit .
43.             <par.qualityValue.MeasureUnit> uomvocab:derivesFrom ?Units .
44.             <par.qualityValue.MeasureUnit> uomvocab:modifierPrefix ?prefix1 .
45.             ?prefix1 uomvocab:factor ?Factor1 .
46.             BIND (1 AS ?Factor2)
47.     }
48.     UNION
49.     {       ?par DUL:parametrizes ?qualityValue .
50.             ?qualityValue uomvocab:measuredIn ?Units .
51.             <par.qualityValue.MeasureUnit> rdf:type ?AppParMeasureUnitType .
52.             ?Units rdf:type ?AppParMeasureUnitType.
53.             <par.qualityValue.MeasureUnit> rdf:type uomvocab:SimpleDerivedUnit .
54.             ?Units rdf:type uomvocab:SimpleDerivedUnit .
55.             <par.qualityValue.MeasureUnit> uomvocab:derivesFrom ?BasicUnit .
56.             ?Units uomvocab:derivesFrom ?BasicUnit .
57.             <par.qualityValue.MeasureUnit> uomvocab:modifierPrefix ?prefix1 .
58.             ?Units uomvocab:modifierPrefix ?prefix2 .
59.             ?prefix1 uomvocab:factor ?Factor1 .
60.             ?prefix2 uomvocab:factor ?Factor2 .
61.     }
62.     FILTER( xsd:double(?Factor2)*?Value = xsd:double(?Factor1)*<par.value>)
63. }
```

In the third case (lines 25-35), the measurement unit of the parameter of the application profile is a basic unit (line 29), so it cannot be converted (?*Factor1* set to 1), whereas the measurement unit of the parameter of the offering is a derived unit (line 30) that can be converted to the basic unit (line 32) using a modifier ?*Factor2* (e.g. Giga) (lines 32-33). For example, 20KB will be converted to 20*1024=20480 bytes. The fourth case (lines 37-47) is the exact symmetrical one; the measurement unit of the parameter of the offering is a basic unit (line 42), whereas the measurement unit of the parameter of the application profile is a derived unit (line 41). Finally, the fifth case (lines 49-61) is when both the offering and the application profile have parameters of derived units (lines 53-54). Instead of converting the one unit to the other, we convert both of them (lines 57-58) to the basic unit they originate from (lines 55-56), so that they become comparable. Line 62 at the end is the filtering expression that compares the two converted (or not) values, using the alternative ?$Factor_i$ multipliers.

Max or MinMax Value

Actually the SPARQL template is almost the same with the one presented above for the "Single Value" case. The only difference is the comparison operator in the last FILTER expression (line 68), which now becomes as in **Algorithm 12**.

Algorithm 12. SPARQL template for checking a functional parameter with a max numerical value

```
62.    FILTER( xsd:double(?Factor2)*?Value <= xsd:double(?Factor1)*<par.value>)
```

Min or MaxMin Value

In this case, the upper limit can also be "unbounded". This value is always better than the limit set by the application. If we allow the application requirement to set an "unbounded" limit, then this would only match the "unbounded" value of the offering parameter, nothing else. Therefore, we have to reflect this case on the last FILTER expression (line 68), which is replaced by lines 62-67 in **Algorithm 13**.

Algorithm 13. SPARQL template for checking a functional parameter with a min numerical value

```
62.    FILTER( IF( ?Value ="unbounded",
63.            true,
64.            IF(<par.value>="unbounded",
65.                    false,
66.                    xsd:double(?Factor2)*?Value >= xsd:double(?Factor1)*<par.value>
67.    )))
```

4.2.2 Retrieving Non-Functional Parameters

Non-functional parameters are scored even if the retrieved value of the parameter is outside the limits. Therefore, the final FILTER expression (which was present at the templates for the functional parameters) is not needed. The rest of the SPARQL query is almost the same (**Algorithm 14**). Note here that instead of multiple templates according to the type of the quality value that parametrizes the non-functional parameter,

we have a single one with multiple UNION sections that collect the "correct" value. Furthermore, the template query takes care of the case where there are numerical values that are characterized by units that need to be transformed before compared (e.g. GBytes vs. Mbytes). In this way, the SPARQL query calculates the "correct" offering parameter value to be fed to the scoring function, by using prefixes of Derived Units to transform the value of the offering parameter into the same measurement unit as the application profile parameter (e.g. transform Gbytes to Mbytes).

Compared to the single value case (**Algorithm 11**), the focus lies on lines 1 and 62: In line 62 a single ?*Factor* multiplier is calculated by dividing ?*Factor2* by ?*Factor1*. To better grasp this, consider an application request for 512MB memory and an offering of 1GB memory. The former has ?*Factor1*=2^20, whereas the latter has ?*Factor2*=2^30. The ?*Factor* will be 2^10. Thus, the SPARQL query will return (line 1) to the calling algorithm 1GB*2^10=1024MB as the value of the parameter for the offering, so that the scoring function correctly compares this to the 512MB request.

Notice that both the value of the parameter and the concept instance are returned. This is needed when a single concept has multiple parameters (thus multiple values) of the same type. Furthermore, if there are multiple concepts (of the same type) for the same offering, they will also be returned. So, each value is connected to the concept it belongs to. This is needed later in the scoring, in order to be able to score at the concept level and then select the best concept score for each offering. Note also line 4, with the VALUES construct, which takes into account the list of concepts (<*concept-list*>) that have satisfied the functional parameters. If a concept does not belong to this list, then it cannot be used to provide values for non-functional parameters. However, when a concept does not have any functional parameter, but only non-functional parameters, then <*concept-list*> is empty and the query will not return any concepts and values. In this case, line 4 is omitted. We assume that the calling function takes care of this issue.

Algorithm 14. SPARQL template for retrieving values of non-functional numerical parameters

```
1.  SELECT ( xsd:double(?Factor)*?Value as ?Offering-Value ) ?concept WHERE {
2.          <offering.ID> DUL:satisfies ?groundDescription .
3.          ?groundDescription paas:offers ?concept .
4.          VALUES ?concept { <concept-list> }
5.          ?concept rdf:type <concept.type> .
6.          ?concept DUL:hasParameter ?par .
7.          ?par rdf:type <par.type> .
8.          ?par DUL:hasParameterDataValue ?Value .
9.  …
            Identical to lines 15-61 of Single Value functional parameter (Algorithm 11)
61. …
62.         BIND (?Factor2/?Factor1 AS ?Factor)
63. }
```

4.2.3 Queries for Ordinal Values

In this sub-section we describe two SPARQL queries needed by the scoring function for Ordinal values.

Check order of two ordinal values

The following query (**Algorithm 15**) checks if the first Value (*Value1*) is greater than the second one (*Value2*). The two values must belong to parameters of the same type (lines 2-4) and, of course, they must be different (line 6). Then we check if they both belong to ordinal quality values (lines 7-10), and finally, in order for the first value to be greater than the second one, the first value must follow the second one via the *DUL*:*follows* transitive property (line 14) or the second value must precede the first one via the *DUL*:*precedes* transitive property (line 11).

Algorithm 15. SPARQL template for checking order of ordinal values.

```
1.  ASK {
2.          ?par1 DUL:hasParameterDataValue <Value1> .
3.          ?par2 DUL:hasParameterDataValue <Value2> .
4.          ?par1 rdf:type ?par-type .
5.          ?par2 rdf:type ?par-type .
6.          FILTER ( <Value1> != <Value2> )
7.          ?par1 DUL:parametrizes ?QualityValue1 .
8.          ?QualityValue1 rdf:type paas:OrdinalValue .
9.          ?par2 DUL:parametrizes ?QualityValue2 .
10.         ?QualityValue2 rdf:type paas:OrdinalValue .
11.         {       ?QualityValue2 DUL:precedes+ ?QualityValue1 .
12.         }
13.         UNION
14.         {       ?QualityValue1 DUL:follows+ ?QualityValue2 .
15.         }
16. }
```

Count distance between two ordinal values

The following query (**Algorithm 16**) counts the distance (hops) between the first Value (*Value1*) and the second value (*Value2*). The difference between the current and the previous queries is from line 11 onwards. So, instead of just checking if the two values are connected through a *DUL*:*precedes* or*DUL*:*follows* path, we instantiate with the ?*mid* variable all the intermediate nodes of the path and in line 1 we return the total number of these intermediate nodes using the *COUNT* aggregate function.

Algorithm 16. SPARQL template for counting the distance between two ordinal values

```
1.  SELECT (count(?mid) AS ?diff) {
2.  …
            Identical to lines 2-10 of Algorithm 15
10. …
11.         {       ?QualityValue2 DUL:precedes* ?mid .
12.                 ?mid DUL:precedes+ ?QualityValue1 .
13.         }
```

```
14.         UNION
15.         {       ?QualityValue1 DUL:follows* ?mid .
16.                 ?mid DUL:follows+ ?QualityValue2 .
17.         }
18. }
```

4.3 Matchmaking Examples

In this section, we give some examples that better illustrate the matchmaking and scoring procedures of our algorithm. The first offering (**Figure 9**) has java 1.6.0 as programming environment and two different options for database: a MySQL database with up to 1 GB size and a MySQL database with up to 5 GB storage. The second offering (with a similar structure) has java 1.4.0 as programming environment, a MySQL database with 10 GB storage and a MongoDB with 15 GB storage.

Application 1 - SQL type database and java 1.6.0 (as functional): We assume that we have an application profile which requires java 1.6.0 and an SQL database. Our algorithm first checks which offerings have concepts with 'SQLDatabase' as type and 'SQLDatabase' as ServiceType parameter. Afterwards, the algorithm checks the programming environment. For every offering's concept check, first, if the type is of Programming environment type and then check if it has parameter 'LanguageName' and if that parameter has a value equal to 'java'. For every concept that passes this check, if it has 'LanguageVersion' we also need to confirm that this parameter has a value of '1.6.0'. When the algorithm terminates, only the first offering has passed, with 'java' and with '1.6.0' as its version.

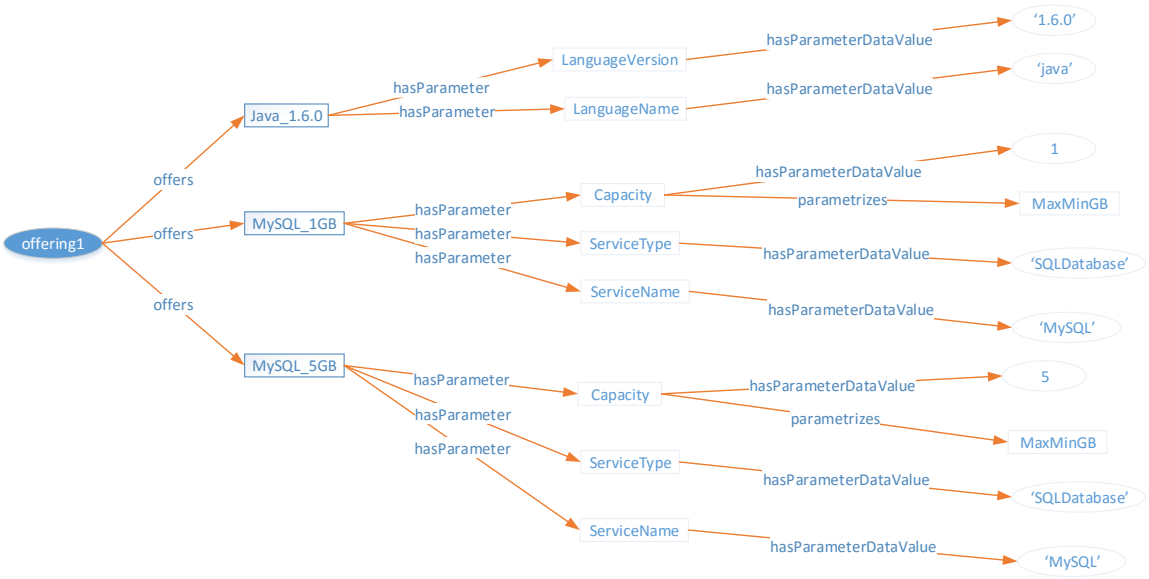

Figure 9. Example 1 of an offering.

Application 2 - MySQL database with 7GB (as storage) and java (as functional): We now have an application that has a MySQL database with 7GB as a first requirement. So, the algorithm for every offering checks

every concept that has 'SQLDatabase' as type, 'MySQL' as service name and if it has a capacity of 7GB or more. Only instance 'MySQL_10GB' has these attributes; thus only offering 2 passes that test. Afterwards, the algorithm checks all remaining offerings for a programming environment concept that has 'java' as LanguageName value. When the algorithm finishes, only offering 2 has passed.

Application 3 - MySQL database (functional) with 1GB as storage (non-functional): In this case the user only needs a MySQL database. If an offering indeed has a MySQL database, then we would like to give it a higher score if the storage is larger than 1GB. Our algorithm first checks all concepts of every offering if they contain the functional parameters ('MySQL' as service name and 'SQLDatabase' as type). For every concept that contains the functional parameters, it checks the value for non-functional parameters and assigns a score to the concept. In this example, concepts 'MySQL_1GB', 'MySQL_5GB', 'MySQL_10GB' all have functional parameters and are scored 0.0, 0.44, and 1, respectively. If an offering has more than one concept that passes the functional parameter threshold, we assign the maximum score of these concepts to the offering. Thus, offering 1 scored 0.44 and offering 2 scored 1.

We now present some scoring results of the algorithm. We tested one parameter with different offering values and the same requirement. **Table 1** (columns 2-4) and **Figure 10**-a show the results of different values of RAM capacity. The offerings' values vary between 512MB to 6 GB. Columns 5-7 of **Table 1** and **Figure 10**-b illustrate the results of different values of RAM capacity, again, but an offering has "unbounded" as value.

Table 1. Values of scoring functions with and without "unbounded" value.

RAM capacity (GB)	*without "unbounded" value*			*with "unbounded" value*		
	linear	**sublinear**	**superlinear**	**linear**	**sublinear**	**superlinear**
0.5	0	0	0	0	0	0
1	0.0909	0.019	0.36	0.07	0.015	0.31
2	0.2727	0.06	0.74	0.22	0.05	0.67
3	0.4545	0.1212	0.89	0.37	0.09	0.84
4	0.6363	0.2	0.95	0.52	0.14	0.92
5	0.8181	0.34	0.98	0.66	0.22	0.96
6	1	1	1	0.81	0.34	0.98

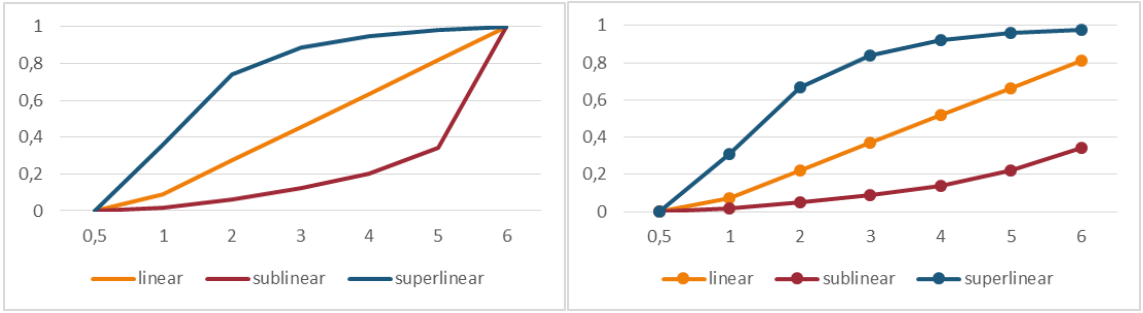

Figure 10. Scoring functions (a) without "unbounded" value and (B) with "unbounded" value.

Finally, we change the quality value of parameter from Max to Min/MaxMin, thus, the minimum value is better. The table below presents the results of the execution.

Table 2. Inverse values (1-x) of scoring functions.

RAM capacity (GB)	linear	superlinear	sublinear
0.5	1	0.99	1
1	0.9	0.98	0.47
2	0.72	0.97	0.25
3	0.54	0.93	0.15
4	0.36	0.83	0.09
5	0.18	0.59	0.04
6	0	0	0

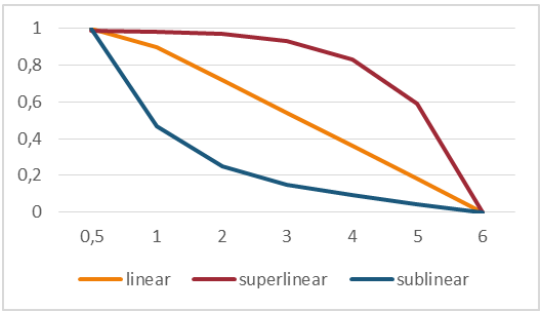

Figure 11. Inverse scoring functions.

4.4 Interaction with the Persistence Layer

The Persistence (or Repository) Layer is used in order to persist the various PaaSPort data models that are mapped to the semantic model and other entities that are needed for the proper function of PaaSPort Marketplace, while also offering appropriate search and discovery interfaces that allow the usage of persisted information from other components (PaaSport Consortium, 2015). The main component of the persistence layer is a Relational database that is used to store the data that are necessary for the operation of the platform. The repository contains: a) the semantic profiles of the PaaS offerings advertised in the Marketplace; and b) the semantic profiles of the deployed business software applications.

In order to expose the PaaS offerings stored in the repository layer to the recommendation layer for the purposes of semantic matchmaking and ranking, we use the D2RQ Platform (D2RQ Platform, 2012), which is a system for accessing relational databases as virtual, read-only RDF graphs. It offers RDF-based access to the content of relational databases without having to replicate it into an RDF store. The D2RQ Platform provides a language for mapping relational database schemas to RDF vocabularies and OWL ontologies, through a mapping document that defines a virtual RDF graph that contains information from the database. In PaaSport we use the D2RQ mapping language in order to export the offering profiles from the relational database of the persistence layer to an RDF format (PaaSport Consortium, 2014d), so that they can be used

by the matchmaking and ranking algorithm of the recommendation layer. Every time that a PaaS provider inserts a new offering instance in the database, a script is responsible to recreate the PaaSport ontology file.

The Recommendation layer interacts with the persistence layer, through the matchmaking algorithm, to:

1. retrieve the PaaS offerings stored in the PaaSport Marketplace database
2. get the application requirements that the DevOps engineer has posted through the UI

Figure 12 shows the workflow of data from the persistence layer to the recommendation layer, concerning the inputs needed for the matchmaking/recommendation algorithm. The PaaS offerings are stored in the relational database of the persistence layer and they are mapped to RDF data, using the concepts and properties of the Semantic Models, using the D2RQ platform presented above. Then the offerings are fed into the matchmaking algorithm. After that, the application requirements are queried from the persistence layer and they are used to construct an application object that is also used as input of the matchmaking/recommendation algorithm, which then proceeds as already described above.

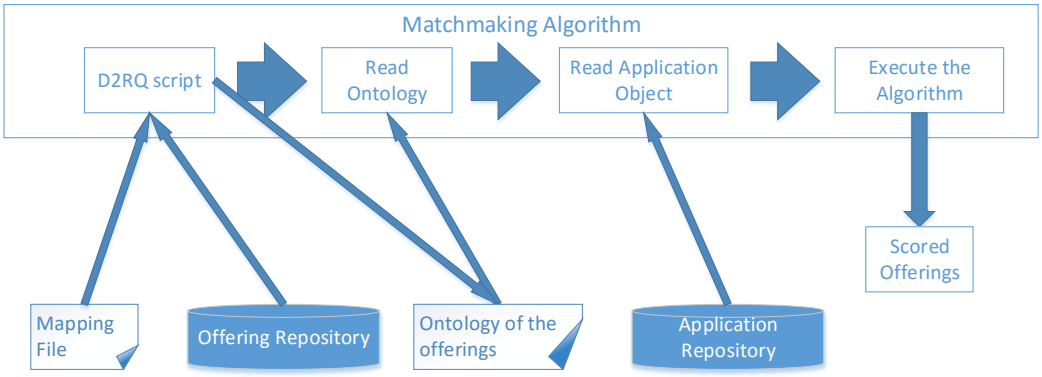

Figure 12. Workflow of Data for the Recommendation layer

5 Evaluation

In order to evaluate the scalability of the recommendation algorithm we have set up an experiment where we are generating three types of PaaS offerings multiple times and we measure the response time of the algorithm. **Table 3** shows the concepts and the parameters of the three offering types. The "small offering" type has the "worse" values for the various numerical range quality values, such as bandwidth, storage capacity, latency, etc., whereas the "large offering type" has the "best" values. These offerings reside in a file and are loaded by Jena in main-memory, so we are just testing the algorithm performance and not the performance of the whole marketplace. However, since in the actual marketplace deployment the offerings will reside in a file that will be re-generated each time offerings are added to the repository, our experiments are still valid regarding the algorithm performance in the actual PaaSport marketplace.

Table 3. Concepts and parameters for the three offering types of the evaluation.

Concept ID	Concept Type	Concept Parameters	Value Type	Small Value	Medium Value	Large Value
Network	Network	NetworkBandwidth	Min	1000	1500	2000
		NetworkLatency	Max	400	200	100
		NumberOfSSLEndpoints	MaxMin	1	2	4
PHP 5.3	Programming-Language	LanguageName	SingleValue	PHP	PHP	PHP
		Version	SingleValue	5.3	5.3	5.3
PHP 5.4	Programming-Language	LanguageName	SingleValue	PHP	PHP	PHP
		Version	SingleValue	5.4	5.4	5.4
python 2.6	Programming-Language	LanguageName	SingleValue	python	python	python
		Version	SingleValue	2.6	2.6	2.6
python 3.0	Programming-Language	LanguageName	SingleValue	python	python	python
		Version	SingleValue	3.0	3.0	3.0
dJango	Programming-Framework	LanguageName	SingleValue	python	python	python
		LanguageVersion	SingleValue	2.6	2.6	2.6
		FrameworkName	SingleValue	dJango	dJango	dJango
		FrameworkVersion	SingleValue	1.7	1.7	1.7
java 7	Programming-Language	LanguageName	SingleValue	java	java	java
		Version	SingleValue	7.0	7.0	7.0
java 8	Programming-Language	LanguageName	SingleValue	java	java	java
		Version	SingleValue	8.0	8.0	8.0
QoS	QoS	Latency	Max	400	200	100
		Uptime	Min	95	98	100
Mongodb	NoSQLDatabase	ServiceName	SingleValue	mongoDB	mongoDB	mongoDB
		Version	SingleValue	2.6	2.6	2.6
		StorageCapacity	MaxMin	1	2	4
MySQL	SQLDatabase	ServiceName	SingleValue	mySQL	mySQL	mySQL
		Version	SingleValue	5.0	5.0	5.0
		StorageCapacity	MaxMin	1	2	4
postgresSQL	SQLDatabase	ServiceName	SingleValue	postgresSQL	postgresSQL	postgresSQL
		Version	SingleValue	5.0	5.0	5.0
		StorageCapacity	MaxMin	1	2	4
Processing	Processing	Frequency	Min	400	800	4000
		MemoryCapacity	MaxMin	1	2	4
		NumberOfCores	MaxMin	1	2	4
		ScalingAvailability	SingleValue	TRUE	TRUE	TRUE
Storage	Storage	StorageCapacity	MaxMin	1	2	4

We have repeated the experiment with various application requirements, both functional and non-functional (see **Table 4**). The storage capacity requirement for the container of the platform (next-to-last line of **Table 4**) is a functional requirement and differs from 1 GB to 5 GB, in order to test the scalability of the algorithm for various percentages of PaaS offerings that satisfy the functional requirements and pass from the first loop of the recommendation algorithm (see section 4.1) that checks functional requirements to the second loop that ranks non-functional requirements for the offerings that satisfy the functional requirements. Therefore, since the three types of PaaS offerings have container storage capacity of 1, 2 and 4 GB, this means that by differing the corresponding application functional requirement, the number of offerings that pass from the first loop to the second vary from 100% (when the requirement is for 1 GB storage capacity), to 66% (2 GB), 33% (3 GB), 0% (5 GB).

Table 4. Application requirements for the algorithm evaluation

PaaS Concepts	PaaS concept Parameters		
Processing	number of cores = 2 (non-functional)	frequency = 1000 Hz (non-functional)	
QoS	latency = 300 ms (non-functional)	uptime = 98% (non-functional)	
ProgrammingLanguage	name = 'PHP' (functional)	version = 5.3 (functional)	
Storage	storageCapacity = 1 | 2 | 3 | 5 GB (functional)		
NoSQLDatabase	name = 'mongodb' (functional)	version = 2.6 (functional)	storageCapacity = 2GB (non-functional)

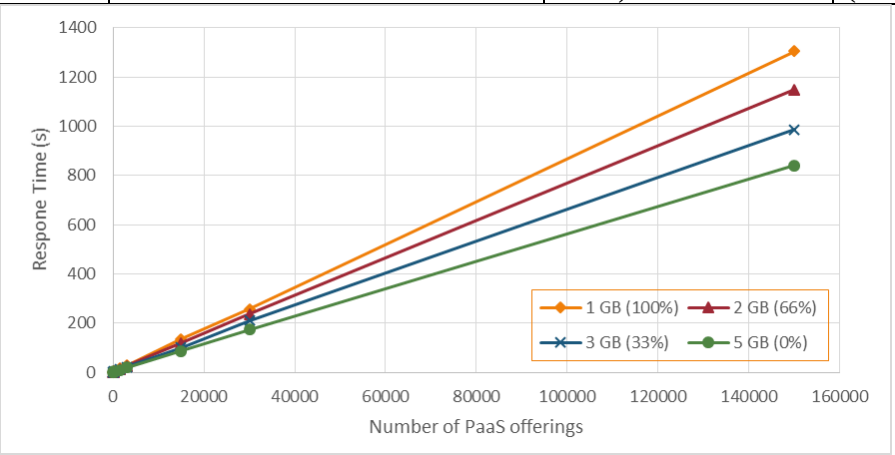

Figure 13. Recommendation algorithm scalability with the number of PaaS offerings

Figure 13 shows the response time of the recommendation algorithm vs. the number of PaaS offerings for the four different application requirements of **Table 4**. The experiments run on a Windows PC with 8 core CPU at 4GHz, 16 GB RAM, SSD. The number of offerings varies from 30 to 150,000. In a reasonable real-world setting that number would be in the order of 100-1000. It is clear that the algorithm is linear to the number of offerings, verified by a regression analysis shown in **Table 5**, where it can be seen that the processing time per offering is about 8,7 msec, giving actual executions times between 2-10 sec in a real-world setting. Furthermore, the response time of the algorithm also depends on the number of matched offerings, i.e. the ones that satisfy the functional requirements and pass to the ranking phase of the non-functional requirements. The less the offerings pass to the second phase, the fastest the algorithm, since the second loop runs for fewer offerings. In the extreme case of 5 GB, no PaaS offering passes to the ranking phase.

Table 5. Regression analysis for response time over number of offerings for various percentages of matched offers

Percentage of matched offerings	Linear function	R^2
100%	y = 0,0087x + 1,6137	1
66%	y = 0,0076x + 3,0289	1
33%	y = 0,0066x + 3,1742	0,9999
0%	y = 0,0056x + 2,3519	1

In order to estimate the impact of non-functional parameters on the running time of the recommendation algorithm we have repeated the experiment with only the storage capacity requirement (**Table 4**). Specifically, we have run the recommendation algorithm with various numbers of offering when storage capacity is a) the only non-functional application requirement, b) the only functional application requirement, and c) the application requirement that is both functional and non-functional. In cases (b) and (c), all offerings pass the functional requirement. Results are shown in **Figure 14**, where it is evident that the execution time of functional and non-functional parameters is almost identical. When both are present their execution time adds up (almost doubles). This is due to the fact that the complexity of both parts of the recommendation algorithm is identical and there is a single SPARQL query of constant time in the inner loop of both parts.

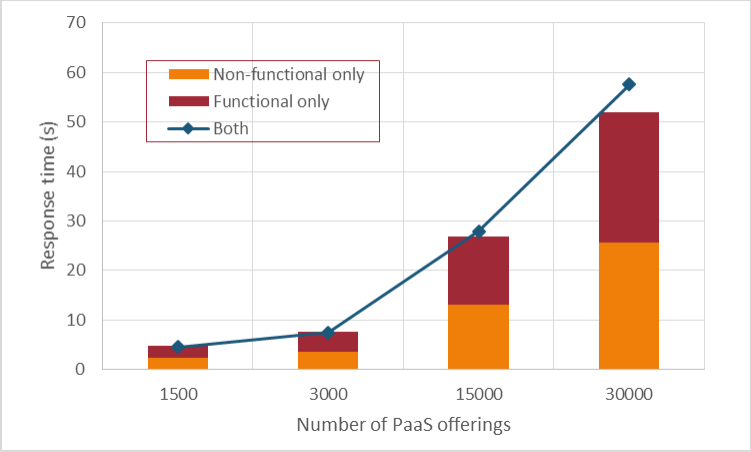

Figure 14. Testing the impact of functional vs. non-functional parameters.

Table 6. Incremental application requirements for testing the impact of the number of parameters.

Experiment	PaaS Concepts	Total Number of Parameters
1	Processing	2
2	Processing + QoS	4
3	Processing + QoS + ProgrammingLanguage	6
4	Processing + QoS + ProgrammingLanguage + Storage (1GB)	7
5	Processing + QoS + ProgrammingLanguage + Storage (1GB) + NoSQLDatabase	10

Furthermore, we have tested the impact of the number of parameters on the execution time. Notice that parameters are associated to concepts and the recommendation algorithm has two loops: one regarding concepts and one regarding parameters of a certain concept. However, what really matters in terms of execution time is how many SPARQL queries are executed; this depends on the total number of parameters since there is one SPARQL query per parameter. We are using the application requirements of **Table 4** in that order. At experiment 1 we are using the concept/parameters of the first line; at each consecutive experiment *i* we

are using accumulatively the concepts/parameters up to line *i* (**Table 6**). For the container storage functional requirement we have used 1GB, meaning that 100% of the offerings are selected and ranked.

We have repeated the experiment for four different number of offerings (1,5K - 30K). Results (**Figure 15**) show that the algorithm scales up also linearly with the number of the application requirement parameters. This is also verified by a regression analysis shown in **Table 7**. The lower values of R^2 are due to the fact that not all application requirements are similar; some of them are functional and some non-functional. The previous experiment (**Figure 14**) has shown that their execution times are similar but not identical.

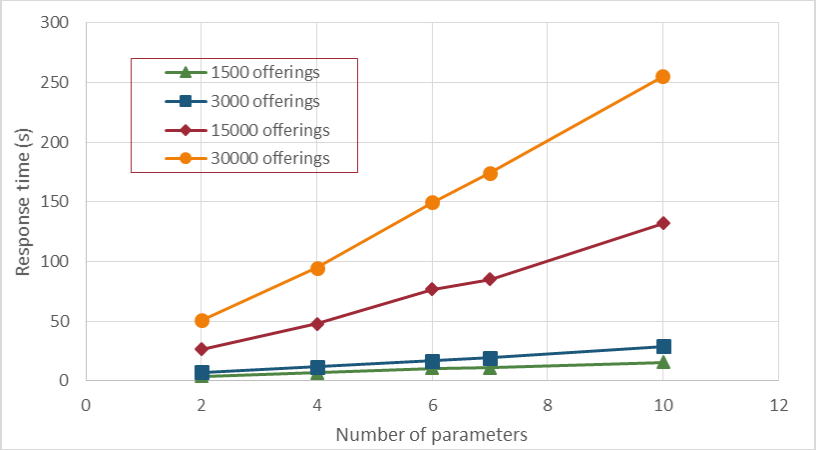

Figure 15. Recommendation algorithm scalability with the number of application requirements

Table 7. Regression analysis for response time over number of requirements for various numbers of offerings

Number of offerings	Linear function	R^2
1.500	y = 1,4362x + 1,1151	0,9959
3.000	y = 2,7043x + 1,0353	0,9935
15.000	y = 13,23x - 3,1153	0,9937
30.000	y = 25,695x - 4,2433	0,9986

An obvious question that arises after the finding that the response time is linear to the number of offerings and the number of parameters is how the response time depends on these two parameters concurrently, i.e. what is the shape of the function $T(N,R)$, where N is the number of offerings and R is the number of parameters. The analysis in section 4.1 suggest that the time complexity of the recommendation algorithm is $O(N*R)$, which suggests that its 3D plot should be a hyperbolic paraboloid surface (Weisstein, 2016). In **Figure 16** we have plotted the response time according to the number of offerings and requirement parameters in a surface. We have used the curve fitting toolbox of Matlab to check whether this surface is a hyperbolic paraboloid indeed. The general equation that time complexity $O(N*R)$ suggest is f(x,y) = a + b*x + c*y + d*x*y. The curve fitting results are shown in **Table 8** and they confirm that time complexity of the

recommendation algorithm is O(*N***R*). The coefficient of the N*R term indicates that each iteration of the inner loop (SPARQL query) lasts less than a millisecond (0.0008168 s).

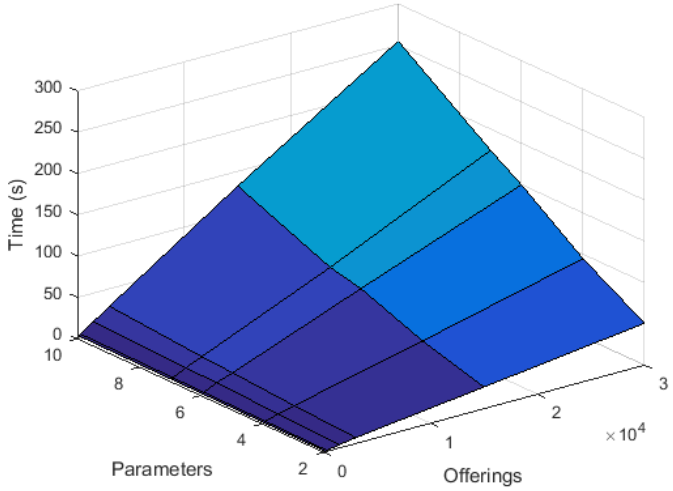

Figure 16. Response time surface plot according to number of offerings and number of requirements

Table 8. Results of curve fitting

```
General model:
     f(N,R) = a + b*N + c*R + d*N*R
Coefficients (with 95% confidence bounds):
       a =      0.6921  (-0.9042, 2.288)
       b =   2.756e-05  (-8.844e-05, 0.0001435)
       c =      0.1779  (-0.07136, 0.4272)
       d =   0.0008168  (0.0007986, 0.0008349)

Goodness of fit:
  SSE: 53.54
  R-square: 0.9995
  Adjusted R-square: 0.9995
  RMSE: 1.435
```

Finally, we experimented with the impact of the ordering of the functional requirement that filters out offerings. As it was discussed in section 4.1, if an offering violates a single functional parameter it is ruled out; therefore, the sooner a non-selectable offering is ruled out the better because the rest of the requirements will not be checked for it, saving time. In this experiment we have used the container storage requirement of 3GB, meaning that 33% of the total offerings are selected. The experiment was performed on checking the storage functional requirement in the first position up to the last position out of the 5 PaaS concepts (**Table 4**) and for different number of offerings (1,5K up to 30K). **Figure 17** shows the results. Notice that in the horizontal axis we have the number of parameters checked (for the offerings that are filtered out due to the violation of the functional requirement), which depends on the order that the functional requirement is checked. If it is checked first, then it will be the only parameter checked for the 66% of the non-selected offerings. If it is checked second, then the 2 parameters of the first concept (see **Table 4**) along with the

storage parameter (3 in total) will be checked for the 66% of the non-selected offerings, and so on so forth. **Table 9** summarizes the total number of parameters checked for the non-selected offerings, according to the position that the storage parameter is checked. It is evident that the algorithm scales up also linearly with the number of the application requirement parameters. This is also verified by a regression analysis shown in **Table 10**. The lower values of R^2 are due to the fact that not all application requirements are similar; some of them are functional and some non-functional. The previous experiment (**Figure 14**) has shown that their execution times are similar but not identical. What can be concluded is that a wise selection of the order in which the functional requirement parameters are checked can save up to 58% of the total execution time (see **Figure 17** for 30K offerings, first vs. last point of the graph). Of course, for larger numbers of offerings and requirement parameters the savings will be even higher.

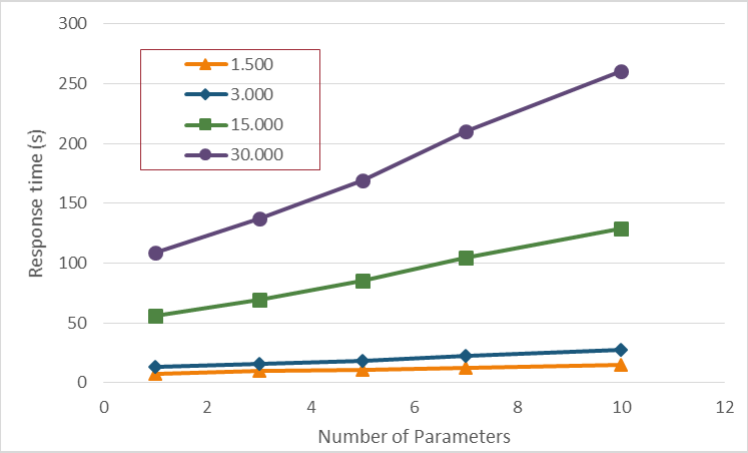

Figure 17. Testing the impact of the order of checking a functional parameter.

Table 9. Number of parameters checked for non-selected offerings according to storage parameter position.

Position of parameter	Total Number of Parameters checked
1	1
2	3
3	5
4	7
5	10

Table 10. Regression analysis for response time over number of requirements for various numbers of offerings.

Number of offerings	Linear function	R^2
1.500	y = 0,8581x + 6,4376	0,9925
3.000	y = 1,6764x + 10,489	0,9918
15.000	y = 8,2518x + 45,761	0,9974
30.000	y = 17,16x + 87,935	0,9972

6 Conclusions and Future Work

The PaaSport project aims to avoid the cloud provider lock-in problem of software SMEs, by enabling platform provider SMEs to roll out semantically interoperable PaaS offerings and facilitating the former to deploy business applications on the best-matching Cloud PaaS or to seamlessly migrate these applications on demand. To this end, PaaSport combined Cloud PaaS technologies with lightweight semantics in order to specify and deliver a thin, non-intrusive Cloud-broker, in the form of a Cloud PaaS Marketplace.

In this paper we have presented the semantical aspects of the PaaSport cloud broker / marketplace. More specifically, we have developed an OWL ontology for representing the necessary PaaS concepts and attributes that are used to semantically annotate a) PaaS offerings, and b) profiles of applications to be deployed on one of the above cloud platforms through a Cloud Broker called PaaSport. The PaaSport ontology has been defined as an extension of the DOLCE+DnS ontology design pattern (DUL) (Gangemi & Mika, 2003). This offers extensibility, since both PaaS concepts and parameters are defined as classes, so extending the ontology with new concepts requires just to extend class hierarchies without adding ontology properties.

On top of the PaaS ontology, we have developed a semantic matchmaking and ranking algorithm for recommending the best-matching Cloud PaaS offering to the application developer, which uses SPARQL queries for retrieving relevant data from the semantic repository. The recommendation algorithm first rules out inconsistent offerings (matchmaking), taking into account the functional requirements of the application profile, and then scores offerings and ranks them accordingly (ranking), considering the non-functional requirements of the application. Due to the fact that application requirements and PaaS offerings share the same vocabulary for PaaS concepts and parameters, the recommendation algorithm seamlessly matches requirements to offerings both syntactically and semantically. The recommendation algorithm scales-up linearly with the number N of instances and the number R of requirement parameters, with an overall time complexity of $O(N*R)$. One of the main advantages of the algorithm is that it extensible because it is agnostic to domain specific concepts and parameters, due to the extensible nature of the PaaS ontology.

Future development plans for the recommendation algorithm, include exploring potential performance improvements through parallelization, since the loop over the offerings in the matchmaking part of the functional requirements can be broken down into concurrent threads of execution. Another interesting direction would be a sophisticated tuner that selects the best order to check the requirements in order to save time by filtering out early offerings that are bound to violate some functional requirement. Of course, the execution time of the tuner should be far less than the execution time of the actual recommendation algorithm. Finally, since the recommendation algorithm is mostly based on SPARQL query templates, the integration of the algorithm within the ontology as SPIN / SPARQL rules would offer transparency, modifiability, extensibility and portability.

Acknowledgments

This work is fully funded by the EU FP7-SME-2013-2-605193 PaaSport project. The authors would like also to thank our project partners Giannis Ledakis, Andreas Papadopoulos, Demetris Trihinas, George Pallis, Gerald Hübsch, Fatemeh Ahmadi Zeleti, Sonya Abbas, Islam Hassan, and Erdem Gulgener, for their valuable comments.

References

[1] 4CaaSt FP7 project. (2013). http://www.4caast.eu Accessed 20.07.16

[2] Andrikopoulos, V., Binz, T., Leymann, F., & Strauch, S. (2013). How to adapt applications for the Cloud environment: Challenges and solutions in migrating applications to the Cloud. *Computing, 95(6),* 493-535.

[3] Andrikopoulos, V., Zhe, S., & Leymann, F. (2013). Supporting the Migration of Applications to the Cloud through a Decision Support System. In *Proceedings IEEE Sixth Int. Conf. on Cloud Computing (CLOUD 2013)* (pp. 565-572).

[4] Androcec, D., Vrcek, N., & Kungas, P. (2015). Service-Level Interoperability Issues of Platform as a Service. In *Proceedings IEEE World Congress on Services (SERVICES 2015)* (pp. 349-356).

[5] Apache Jena. (2016). https://jena.apache.org/ Accessed 20.07.16

[6] Borenstein, N., & Blake, J. (2011). Cloud Computing Standards: Where's the Beef? *IEEE Internet Computing, 15(3)*, 74-78.

[7] Bosi, F., Ravagli, F., Laudizio, V., Porwol, L., & Zeginis, D. (2012). SOA layer software components v2, Version 1.0. WP 4 Cloud4SOA SOA Layer, 26/05/2012. (not publicly available).

[8] Carvalho, L., Mahowald, R. P., McGrath, B., Fleming, M., & Hilwa, A. (2015). Worldwide Competitive Public Cloud Platform as a Service Forecast, 2015–2019. Jul 2015. Doc # 257391. Market Forecast. https://www.idc.com/getdoc.jsp?containerId=257391 Accessed 20.07.16

[9] Cloud4SOA FP7 project. (2012). http://www.cloud4soa.com Accessed 20.07.16

[10] Columbus, L. (2013). 451 Research: Platform-as-a-Service (PaaS) Fastest Growing Area of Cloud Computing. http://www.forbes.com/sites/louiscolumbus/2013/08/20/451-research-platform-as-a-service-paas-fastest-growing-area-of-cloud-computing/ Accessed 20.07.16

[11] D2RQ Platform. (2012). http://d2rq.org/ Accessed 20.07.16

[12] European Commission. (2012). Unleashing the Potential of Cloud Computing in Europe. *Communication from the Commission to the European Parliament, the Council, the European Economic and Social Committee and the Committee of the Regions*. Brussels, COM(2012) 529 final, 27/9/2012. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=COM:2012:0529:FIN:EN:PDF Accessed 20.07.16

[13] Gagliardi, F., & Muscella, S. (2010). Cloud computing: data confidentiality and interoperability challenges. In: *Principles, Systems and Applications (Computer Communications and Networks)* (pp. 257-270). London: Springer.

[14] Gangemi, A., & Mika, P. (2003). Understanding the semantic web through descriptions and situations. In: *Proc. Int. Conf. on Ontologies, Databases and Applications of Semantics* (pp. 689-706).

[15] Gardner, D. (2010). Cloud computing's ultimate value depends on open PaaS models to avoid applications and data lock-in. http://www.zdnet.com/article/cloud-computings-ultimate-value-depends-on-open-paas-models-to-avoid-applications-and-data-lock-in/ Accessed 20.07.16

[16] Gupta, S., Muntes-Mulero, V., Matthews, P., Dominiak, J., Omerovic, A., Aranda, J., & Seycek, S. (2015). Risk-driven Framework for Decision Support in Cloud Service Selection. In *Proc. 15th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CC-GRID 2015)* (pp. 545–554).

[17] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA Data Mining Software: An Update. *SIGKDD Explorations, 11(1)*.

[18] Kamateri, E., Loutas, N., Zeginis, N., Ahtes, J., D'Andria, F., Bocconi, S., … Tarabanis, K. (2013). Cloud4SOA: A semantic-interoperability PaaS solution for multi-Cloud platform management and portability. In *Service-Oriented and Cloud Computing* (pp. 64-78). *Lecture Notes in Computer Science, Vol. 8135*.

[19] Klusch, M., & Kapahnke, P. (2008). Semantic Web Service Selection with SAWSDL-MX. In *Proc. 2nd International Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web (SMRR). CEUR Proceedings, vol. 416*.

[20] Klusch, M., & Kapahnke, P. (2010). iSeM: Approximated Reasoning for Adaptive Hybrid Selection of Semantic Services. In *Proc. European Semantic Web Conference* (pp. 30-44).

[21] Klusch, M., Kapahnke, P., & Zinnikus, I. (2009). SAWSDL-MX2: A Machine-Learning Approach for Integrating Semantic Web Service Matchmaking Variants. In *Proc. IEEE International Conference on Web Services* (pp. 335-342).

[22] Lampe, U., & Schulte, S. (2012). Self-Adaptive Semantic Matchmaking Using COV4SWS.KOM and LOG4SWS.KOM. In *Semantic Web Services* (pp. 141–157). Springer.

[23] Lund, M. S., Solhaug, B., & Stolen K. (2011). *Model-Driven Risk Analysis: The CORAS Approach*. Springer.

[24] Meditskos G. & Bassiliades N. (2010). Structural and role-oriented web service discovery with taxonomies in OWL-S. *IEEE Transactions on Knowledge and Data Engineering, 22(2)*, 278–290.

[25] MODAClouds FP7 project. (2016). http://www.modaclouds.eu Accessed 20.07.16

[26] OASIS CAMP TC. (2014). Cloud Application Management for Platforms Version 1.1. In J. Durand, A. Otto, G. Pilz, & T. Rutt (Eds.), *OASIS Committee Specification 01*. http://docs.oasis-open.org/camp/camp-spec/v1.1/cs01/camp-spec-v1.1-cs01.html Accessed 20.07.16

[27] PaaSage FP7 project. (2016). http://www.paasage.eu Accessed 20.07.16

[28] PaaSport Consortium. (2014a). Deliverable 1.1: PaaSport Requirements Analysis Report, November 2014. http://paasport-project.eu/download/deliverables/paasport_d1.1_finalrelease.pdf Accessed 20.07.16

[29] PaaSport Consortium. (2014b). Deliverable 1.2: PaaSport Reference Architecture, October 2014. http://paasport-project.eu/download/deliverables/paasport_deliverable1.2_architecture_updated_final.pdf Accessed 20.07.16

[30] PaaSport Consortium. (2014c). Deliverable 1.3: PaaSport Semantic Models, November 2014. http://paasport-project.eu/download/deliverables/paasport_deliverable1.3.pdf Accessed 20.07.16

[31] PaaSport Consortium. (2014d). Deliverable 2.1: PaaSport Recommendation Algorithm and Model, November 2014. http://paasport-project.eu/download/deliverables/Paasport_Deliverable_2.1-V1.0-Final.pdf Accessed 20.07.16

[32] PaaSport Consortium. (2015). Deliverable 5.2: PaaSport Marketplace Infrastructure – First Release, May 2015. http://paasport-project.eu/download/deliverables/PaaSport_D5.2_finalrelease.pdf Accessed 20.07.16

[33] PaaSPort FP7 project. (2016). http://paasport-project.eu Accessed 20.07.16

[34] Paolucci M., Kawamura T., Payne T. R., & Sycara K. P. (2002). Semantic Matching of Web Services Capabilities. In I. Horrocks & J. A. Hendler (Eds.), *Proc. 1st International Semantic Web Conference (ISWC '02)* (pp. 333-347). London: Springer.

[35] Quinton, C., Romero, D., & Duchien, L. (2016). SALOON: a platform for selecting and configuring cloud environments. *Software: Practice and Experience*, *46*, 55–78.

[36] Skoutas, D., Sacharidis, D., Simitsis, A., & Sellis, T. K. (2008). Serving the Sky: Discovering and Selecting Semantic Web Services through Dynamic Skyline Queries. *In Proc. Int. Conf. on Semantic Computing (ICSC)* (pp. 222-229).

[37] Walraven, S., Truyen, E., & Joosen, W. (2014). A comparison of PaaS platforms based on a practical case study: Comparing PaaS offerings in light of SaaS development. *Computing, 96(8)*, 669-724.

[38] Wei, D., Wang, T., Wang, J., & Bernstein, A. (2011). SAWSDL-iMatcher: A customizable and effective Semantic Web Service matchmaker. *Web Semantics: Science, Services and Agents on the World Wide Web, 9(4)*, 402–417.

[39] Weisstein, E. W. (2016). Hyperbolic Paraboloid. From *MathWorld--A Wolfram Web Resource*. http://mathworld.wolfram.com/HyperbolicParaboloid.html Accessed 20.07.16

[40] Zapater J. J. S., Escrivá D. M. L., García F. R. S., & Durá J. J. M. (2015). Semantic web service discovery system for road traffic information services. *Expert Systems with Applications, 42(8),* 3833-3842.