EASY: Efficient semAntic Service discoverY in pervasive computing environments with QoS and context support

Sonia Ben Mokhtar, Davy Preuveneers, Nikolaos Georgantas, Valérie Issarny, Yolande Berbers

EASY: **E**fficient Sem**A**ntic **S**ervice Discover**Y** in Pervasive Computing Environments with QoS and Context Support

Sonia Ben Mokhtar Davy Preuveneers Nikolaos Georgantas
Valérie Issarny Yolande Berbers

June 11, 2007

Abstract

Pervasive computing environments are populated with networked software and hardware resources providing various functionalities that are abstracted, thanks to the Service Oriented Architecture paradigm, as services. Within these environments, service discovery enabled by service discovery protocols (SDPs) is a critical functionality for establishing ad hoc associations between service providers and service requesters. Furthermore, the dynamics, the openness and the user-centric vision aimed at by the pervasive computing paradigm call for solutions that enable rich, semantic, context- and QoS-aware service discovery. Although the Semantic Web paradigm envisions to achieve such support, current solutions are hardly deployable in the pervasive environment due to the costly underlying semantic reasoning with ontologies. In this article, we present EASY to support efficient, semantic, context- and QoS-aware service discovery on top of existing SDPs. EASY provides EASY-L, a language for semantic specification of functional and non-functional service properties, as well as EASY-M, a corresponding set of conformance relations. Furthermore, EASY provides solutions to efficiently assess conformance between service capabilities. These solutions are based on an efficient encoding technique, as well as on an efficient organization of service repositories (caches), which enables both fast service advertising and discovery. Experimental results show that the deployment of EASY on top of an existing SDP, namely Ariadne, enhancing it only with slight changes to EASY-Ariadne, enables rich semantic, context- and QoS-aware service discovery, which furthermore performs better than the classical, rigid, syntactic matching, and improves the scalability of Ariadne.

1 Introduction

Pervasive computing [23] envisions the unobtrusive diffusion of computing and networking resources in physical environments, enabling users to access information and computational resources anytime and anywhere, and this in a user-centric way, i.e., where user interaction with the system is intuitive, pleasant and natural. Mobile users

take part in these pervasive environments by carrying around tiny personal devices that integrate seamlessly in the existing infrastructure. Such a setup is highly open and dynamic: pervasive computing systems should support ad hoc deployment and execution, integrating the available hardware and software resources at any given time and place. Incorporation in a larger system is facilitated by rendering such resources into autonomous, networked components.

A recent computing paradigm particularly appropriate for pervasive systems is Service-Oriented Architectures (SOA) [19]. In this architectural style, networked devices and their hosted applications are abstracted as loosely coupled services. Service discovery is an essential function within SOA, as it enables the runtime association to networked services. Three basic roles are identified for service discovery in SOA: (1) *Service provider* is the role assumed by a software entity offering a networked service; (2) *Service requester* is the role of an entity seeking to consume a specific service; (3) *Service repository* is the role of an entity maintaining information on available services and a way to access them. A *service description* formalism or language to describe the capabilities and other non-functional properties (such as quality of service (QoS), security or transactional aspects) complemented with a *service discovery protocol (SDP)* let service providers, requesters and repositories interact with each other. In [34], a classification of academic and industry-supported SDPs, specifically for pervasive environments, is proposed. Service discovery becomes even critical in pervasive environments due to their openness and dynamics and the fact that pervasive software services and potential software clients (assuming the role of service requester) are designed, developed and deployed independently. These concerns raise the following requirements:

- During service discovery, semantics underlying service descriptions and client requests should be matched. In classic service discovery, the matching is based on assessing the conformance of their syntactic interfaces. However, an agreement on a single common syntactic standard is hardly achievable in the open pervasive environment. Thus, higher-level abstractions, independent of the low-level syntactic realizations specific to the SOA technologies in use, should be employed for denoting service semantics.

- Finding a service that exactly matches a client request is rather the exception than the rule in pervasive environments. Thus, matching should be able to identify the degree of conformance between services and clients, and rate services with respect to their suitability for a specific client request.

- The user-centrism of pervasive environments calls for system's awareness of context and QoS. Context [11] is any information that can be used to characterize the situation of the user (location and current activity), the system (available resources) and their interaction. Both context and QoS play a decisive role in enhancing users' experience of the pervasive environment. For establishing a common understanding of such non-functional properties and enabling their matching, semantic abstractions and associated relations assessing the degree of conformance are required – as in the case of functional properties.

A key requirement identified above for service discovery in pervasive environments concerns expressing the semantics of services. A promising approach addressing the semantic modeling of information and functionality comes from the Semantic Web[1] paradigm [7]. Within the Semantic Web, information is enriched with machine-interpretable semantics by referring to a structured vocabulary of terms *(ontology)* representing a specific area of knowledge. Ontology languages, such as the Web Ontology Language (OWL)[2], support formal descriptions and machine reasoning. Building on the Semantic Web, a number of efforts have been conducted in the area of automated semantic Web service discovery, invocation, composition and execution monitoring [29, 28, 16, 30, 25]. However, the employment of semantic technologies and related tools for service discovery in pervasive environments comes with a major handicap: the underlying semantic reasoning is particularly costly in terms of computational resources and not intended for use in highly dynamic and interactive environments. Hence, providing service discovery that is adequate for the demanding pervasive environment and that achieves satisfying performance at the same time for interactivity is still an open issue.

We present in this article our approach for *Efficient semAntic Service discoverY (EASY)* in pervasive environments, which extends the efforts presented in [6, 5] with a number of attractive features. As already mentioned, a large number of SDPs already exist, deployed in various pervasive and mobile computing environments. Our objective is not to propose yet another SDP, but to elaborate a solution for efficient, semantic, context- and QoS-aware service discovery, which can flexibly be deployed on top of existing SDPs. EASY operates at a higher, semantic abstraction level, and is thus independent of the specific underlying SOA technology employed. As part of EASY, we introduce:

- *EASY-Language (EASY-L)*: a language for semantic service description covering both functional and non-functional service characteristics. EASY-L is a simple, generic representation for service functionalities, context properties and QoS properties; and extensible, allowing the addition of new context and QoS dimensions. EASY-L contains only the necessary information to enable service matching.
- *EASY-Matching (EASY-M)*: defines a set of conformance relations and prescribes the way for applying them in order to perform service matching. EASY-M identifies three levels of matching; it further enables assessing the degree of conformance between a service advertisement and a service request, and rating of services with respect to their suitability for a specific request. EASY-M takes into account both functional and non-functional service properties.

With EASY-L and EASY-M, service discovery can be carried out in a service repository to resolve a service request. To ensure efficient EASY service discovery we provide two optimizations:

- By numerically encoding ontologies, we transform the costly semantic reasoning into a simple numeric comparison of codes. Our solution supports incremental

[1] Semantic Web: http://www.w3.org/2001/sw/
[2] OWL: http://www.w3.org/TR/owl-ref/

conflict-free encoding, which allows the freely reuse and extending of existing ontologies; this is an essential requirement for the semantic representation of open pervasive environments and their services. Moreover, our solution proves satisfactory in terms of employed code lengths compared to existing encoding algorithms, which allows saving memory and computational resources.

- We introduce an algorithm for organizing service advertisements in a service repository based on their semantic similarity in order to considerably reduce the number of matchings performed to resolve a service request. Our algorithm consists of two parts: the first concerns inserting a new service advertisement in the organized repository; the second concerns resolving a service request.

The two above optimizations move a great part of the complexity related to semantic service discovery offline, and also improve the performance of the online service registration and resolution of service requests. As such, EASY service discovery becomes highly efficient and applicable for highly interactive pervasive environments. To evaluate the flexibility and scalability of EASY, we elaborate its prototype implementation on top of Ariadne, a semi-distributed SDP for mobile ad hoc networks (MANETs) [22]. We thus enhance Ariadne, which supports syntactic discovery of Web services, into *EASY-Ariadne*, which inherits all features of EASY.

The remainder of this article is structured as follows. In Section 2, we provide a state-of-the-art survey on service discovery enhanced by semantic technologies. Having identified above the high computational cost of semantic technologies as a key issue, we complement this survey with our experimental assessment of this cost, which motivates our approach presented in the next sections. In Section 3, we elaborate on EASY-L and EASY-M, while in Section 4, we discuss our optimizations for efficient EASY service discovery. In Section 5, we present our implementation of EASY on top of Ariadne and our related performance evaluation. Finally, we conclude in Section 6.

2 Service Discovery Enhanced by Semantic Technologies: State of the Art

In this section, we survey current efforts related to semantic service discovery. We first provide an overview of the Semantic Web paradigm and resulting extensions to the semantic description of Web services (Section 2.1). We then discuss related work on service matching based on semantic service descriptions and its application in service discovery (Section 2.2). As pointed out in the previous section, semantic reasoning underlying the semantic Web paradigm involves a high computational cost. To precisely evaluate this cost and its impact on service matching and discovery, we carry out a detailed experimental analysis (Section 2.3). Finally, we survey recent efforts attempting to tackle this problem by introducing optimizations to semantic service matching and discovery (Section 2.4).

2.1 Semantic Web and Semantic Service Description

The World Wide Web contains a huge amount of information, created by multiple organizations, communities and individuals, with different purposes in mind, which makes it hard for Web users (either humans or software agents) to locate the information they are looking for. The Semantic Web vision aims to address this issue by the introduction of semantic annotations in Web pages in order to support effective discovery, data integration and navigation on the Web; this effort especially targets the automation of such tasks, which should be carried out intelligently and efficiently by software agents.

Towards the realization of the Semantic Web objectives, many paradigms and tools are being developed. Ontologies are one of the main building blocks to enable the Semantic Web vision. Ontologies describe structured vocabularies containing useful terms (also called *concepts* or *classes*) for a community that wants to organize and exchange information in a non-ambiguous manner. One of the most widely used languages for specifying ontologies is the Web Ontology Language (OWL). OWL has its formal foundation in Description Logics (DL) [4]; hence, semantics specified in OWL enable semantic reasoning using a DL-reasoner to infer implicit relationships between concepts from the explicit definitions of these concepts in an ontology. Among these relations, the *subsumption* relation allows to relate concepts to more generic concepts in a way similar to inheritance in the object oriented programming model. After subsumption reasoning on an ontology, the resulting hierarchy is referred to as the *classified ontology*.

Ontologies have conveniently been employed in the semantic specification of services. For instance, the latest WSDL (2.0) standard does not only support the use of XML Schema, but also provides standard extensibility features for using, e.g., classes from OWL ontologies to define Web service input and output data types. OWL is further the semantic representation language of choice for the OWL-S[3] proposal, the Web Ontology Language for Web services; OWL-S is a high-level OWL ontology for services. A similar recent proposal for the semantic specification of Web services is the Web Services Modeling Ontology (WSMO)[4], which is specified using the Web Service Modeling Language (WSML). Besides service specification, this ontology provides support for *mediators*, which can resolve mismatches between ontologies or services. The Web Service Semantics - WSDL-S[5] proposal annotates WSDL descriptions with semantics, using references to concepts from, e.g., OWL ontologies, by attaching them to WSDL input, output and fault messages, as well as operations. The First-Order Logic Ontology for Web Services (FLOWS) is a recent proposal for the semantic specification of Web services. It has a well defined semantics in first-order logic enriched with support for Web based technologies (e.g., URIs, XML). FLOWS encloses parts of other languages and standards (e.g., WSMO, OWL-S, PSL (ISO 18629)) and supports a direct mapping to ROWS, another language from the same consortium based on logic programming.

Another effort that focuses more particularly on pervasive services is Amigo-S, which extends OWL-S by integrating features characterizing the heterogeneity and

[3]OWL-S: http://www.daml.org/services/owl-s/1.2/

[4]WSMO: http://www.wsmo.org/

[5]WSDL-S: http://www.w3.org/Submission/WSDL-S/

richness of pervasive environments [3]. For instance, the Amigo-S language enables different service groundings – thus applying to diverse SOAs – and models the related discovery, communication and network protocols. Moreover, Amigo-S models context and QoS properties of services.

As part of our approach in this article, we introduce the EASY-L ontology-based language that has similar characteristics and objectives to Amigo-S (Section 3.1). Thus, we also target independence of the underlying SOA as well as the specification of non-functional properties. Compared to the above related efforts for semantic service specification including Amigo-S, EASY-L captures the set of crosscutting concepts shared by these various languages with additional support for QoS and context properties of services. EASY-L can thus be envisioned as a 'meta-language' that can be easily extended and/or mapped to any of the above languages. Furthermore, unlike Amigo-S, EASY-L does model features of the underlying middleware as it can be applied on top of different middleware infrastructures.

2.2 Semantic Matching for Service Discovery

Based on semantic service description formalisms, such as the ones surveyed in the previous section, a number of research efforts focus on matching between services to compare the suitability of advertised services against a service request. The effectiveness of service matching depends on the expressiveness of service descriptions and on adequate relations assessing the degree of conformance between service descriptions.

A number of research efforts have been conducted in the area of matching semantic Web services based on their signatures. Signature matching deals with the identification of subsumption relationships between the concepts describing inputs and outputs of capabilities [33]. A base algorithm for service signature matching has been proposed by Paolucci *et al.* in [28, 18]. This algorithm allows matching a requested capability, described as a set of *provided inputs and required outputs*, with a number of advertised capabilities, described each as a set of *required inputs and provided outputs*. Inputs and outputs are semantically defined using ontology concepts. The algorithm defines four levels of matching between a provided and a required ontology concept. The four matching levels are:

- *exact*: if the concepts are equivalent or if the required concept is a direct subclass of the provided one
- *plug in*: if the provided concept subsumes the required one,
- *subsumes*: if the required concept subsumes the provided one, and
- *fail*: if there is no subsumption relation between the two concepts.

As part of our approach in this article, we introduce the EASY-M matching (Section 3.2), which adopts (with a slight modification) the same levels of matching for comparing two ontology concepts. Compared to other related efforts for matching service capabilities, EASY-M also supports the matching of non-functional service properties (i.e., QoS and context properties) and provides a means to rate services according to user preferences on the various heterogeneous non-functional properties. The resulting

matching algorithm rates services according to the matching levels evaluated between the concepts used in the service request and those used in the service advertisement. Other solutions based on the above signature matching of semantic Web services have been proposed in the literature [16, 30, 12].

Furthermore, specification matching of software components or, more particularly, semantic Web services, has been studied in the literature [32, 25, 27]. Specification matching deals with matching pre- and post-conditions that describe the functional semantics of components. For instance, in [32], specification matching is performed using theorem proving, i.e., inferring general subsumption relations between logical expressions that specify pre- and post-conditions of components. A more practical way to perform specification matching is to use *query containment* [25, 27]. This is done by modeling both service advertisements and service requests as queries with a set of constraints (e.g., required inputs and outputs are modeled as restrictions on their types). Starting from the specified constraints, the possible values of both queries are evaluated, and possible inclusions between the results of the queries are inferred. Specifically, a query q_1 is contained in q_2 if all the answers of q_1 are included in the answers of q_2.

Whether semantic service matching is performed in terms of service signature or specification, the key issue for efficient matching lies in the performance of the underlying semantic reasoning, as analyzed in the following section.

2.3 Analyzing the Cost of Semantic Service Discovery

In this section we analyze the cost of semantic matching of service capabilities. We primarily focus on evaluating the cost related to DL-reasoning, which is justified by the wide use of such reasoning. To this end, we consider a repository of Web services described using OWL-S. Specifically, we provide an evaluation of the signature matching of service requests against the service advertisements hosted by the repository. The semantic matching between a requested capability and a number of advertised ones is carried out by employing the base matching algorithm by Paolucci *et al.* presented in the previous section.

We have carried out experiments on a notebook with a 1.6 GHz Intel Centrino processor and 512 MB of RAM. Our prototype implementation includes the use of a DL-reasoner to infer the subsumption relationships between concepts. There are various implementations of DL-reasoners; the most popular ones are: Racer[6], FaCT++[7] and Pellet[8]. We provide a performance evaluation of our prototype implementation employing each one of the aforementioned three reasoners in order to assess their impact on the matching process. To this end, we conducted two different kinds of experiments.

Figure 1 shows the results of the first experiment. This experiment gives an overview of the cost of each step of the matching process, i.e.: (1) the time to parse the service advertisement and the service request; (2) the time for the reasoner to load and classify the ontologies involved in the service advertisement and request descriptions; and (3) the time to match the concepts involved in the advertisement and the request, i.e., to

[6]Racer: http://www.sts.tu-harburg.de/ r.f.moeller/racer/
[7]FaCT++: http://owl.man.ac.uk/factplusplus/
[8]Pellet: http://www.mindswap.org/2003/pellet/

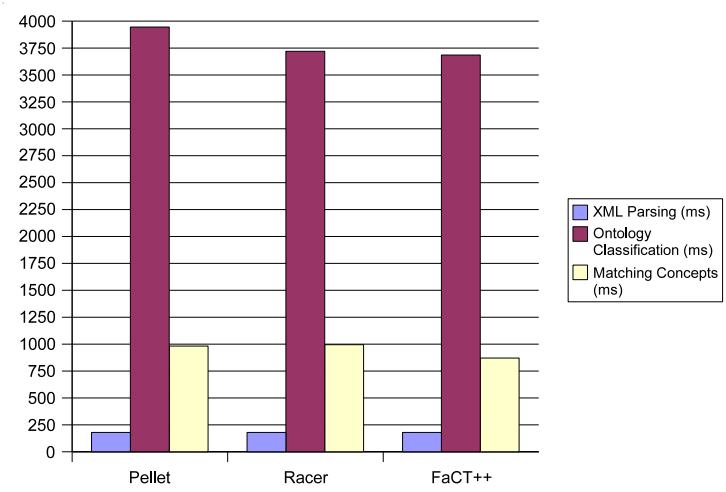


Figure 1: Processing time for each step to match a requested and an advertised capability

assess the relations between these ontology concepts. In this experiment, the service request comprises 7 provided inputs and 3 requested outputs. The ontology used for the experiment is the *Pizza* ontology[9]. This ontology contains 99 OWL classes, 4 datatype properties, 11 object properties, 24 annotation properties and 5 individuals. Results for all three reasoners of the total time of the matching process are in the order of 4 to 5 sec. Furthermore, for all three reasoners, the most expensive phase is the one of loading and classifying the involved ontologies: from 76% to 78%. FaCT++ gives slightly better results than the other two reasoners.

Our second experiment measures the time taken by each reasoner to match the concepts involved in the service request and the service advertisement for an increasing number of concepts. Specifically, we increase the number of concepts involved in the service request from 4 to 14. Figure 2 shows the increasing processing time. We notice that this processing time increases significantly: it grows proportionally to the number of concepts. Again, FaCT++ performs slightly better.

From the above experiments, we conclude that semantic matching of service capabilities is a heavy process. Moreover, the resolution of a single service request in a repository implies carrying out matching between the request and all registered service advertisements, so as to select the advertisement that best fits the request. For a repository containing n service advertisements, the time to match a request with all advertisements is equal to:

$$T_{parse\ request} + n * T_{parse\ advertisement} + T_{classify\ ontologies} + n * T_{match\ concepts}$$

As measured above, for n=1, i.e., one advertisement, and a request including 10 con-

[9]Pizza Ontology: http://www.co-ode.org/ontologies/pizza/

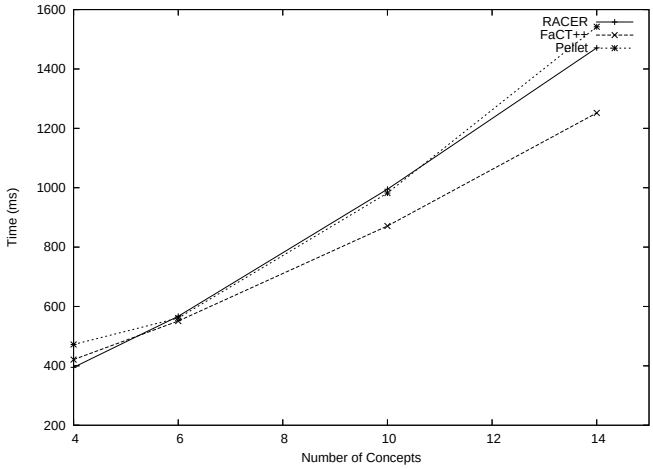


Figure 2: Time to only match concepts

cepts, this time is in the order of 4 to 5 seconds for any of the given reasoners. It is obvious that the cost of semantic reasoning and matching is undesirable for immediate use in the interactive pervasive environment. A number of optimizations need to be introduced to significantly increase efficiency. A first remark towards this direction is that the step of loading and classifying an ontology, as indicated in the above relation, needs to be performed only once. Later on, the classified ontology can be accessed several times to assess relations between concepts that occur more than once in the service requests and advertisements. Therefore, classification should be performed offline, while the rest of the matching steps can be performed online. Moreover, the step of matching concepts, which costs around 25% of the total matching time, has to be performed as many times as there are advertisements in the repository. Thus, optimizations for this step should be investigated. In the following section, we survey efforts towards optimization of semantic service discovery. In Section 4, we elaborate our approach to efficient service discovery.

2.4 Optimizations to Semantic Service Discovery

Several research efforts attempt to optimize the costly semantic service discovery. We distinguish efforts that: numerically encode ontology hierarchies, so as to reduce semantic reasoning to numerical comparison of codes (Section 2.4.1); organize semantic service advertisements in repositories, so as to reduce the number of matchings performed for resolving a service request (Section 2.4.2); combine encoding and organizational techniques (Section 2.4.3).

2.4.1 Encoding the Multiple Inheritance Hierarchy of an Ontology

For encoding ontology hierarchies, various techniques may be sought in other related problem areas, such as the encoding of class hierarchies in object-oriented programming languages. However, ontology encoding also needs to deal with issues typical for knowledge representation, such as support for incremental encoding without causing conflicts for existing codes while achieving efficient matching.

A straightforward constant time matching technique encodes n classes in a hierarchy using a $n \times n$ binary matrix with a 1 on positions (i, j) if class i is an ancestor of class j. Ait-Kaci *et al.* [2] achieved a more compact bottom-up approach, requiring less than n^2 bits, that selects different bit positions for leaf classes and encodes each parent using the OR operation on the children's codes. Caseau [8, 9] and Krall [14] proposed graph coloring based approaches that further improve the compactness of the code, but conflict-free incremental encoding or solving conflicts for new classes in an efficient way is no longer straightforward. Numeric intervals are also used to encode inheritance by ensuring that children have non-overlapping intervals that fall inside the interval of their parent. The *relative numbering* [24] algorithm traverses the tree in post-order to define the range of a class' interval, but it neither supports multiple inheritance nor incremental encoding. To support multiple inheritance, both Agrawal *et al.* [1] and Constantinescu *et al.* [10] add additional intervals to parent classes to correct the interval containment of the children. However, inheritance testing for classes with multiple intervals can become an expensive operation. Zibin *et al.* [35] proposed an algorithm based on relative numbering and PQ-trees that improves the encoding length and test time of the algorithm of Krall *et al.* [14], but the algorithm cannot be easily modified for incremental encoding. As such, the quest for a compact representation that can be easily extended without introducing conflicts in the existing codes and that enables efficient matching, motivates the introduction of a new ontology encoding scheme.

2.4.2 Organizing Service Descriptions

For organizing semantic service advertisements in repositories, solutions may be sought in service classifications. The OWL-S specification provides the means for defining hierarchies of service descriptions called *profile hierarchies*. These hierarchies are similar to the object-oriented inheritance hierarchies. For instance, when a new service profile is defined, it may be specified as a subclass of an existing profile class. This allows the new service to inherit all the properties of all the classes specified in its super-hierarchy of classes. While this approach allows the classification of service profiles according to the classes from which they inherit, it does not allow considering possible relations between service profiles that do not have the same common set of properties but still provide similar functional features. Service classification can also be based on the service category using existing taxonomies such as NAICS[10] or UNSPSC[11]. However, service categories alone do not give enough information about the service functionality.

[10] NAICS taxonomy: http://www.census.gov/epcd/www/naics.html
[11] UNSPSC taxonomy: http://www.unspsc.org/

2.4.3 Combined Techniques for Efficient Service Discovery

Several efforts towards efficient semantic service discovery have been proposed in the literature [10, 26, 31]. In [10] the authors propose to numerically encode service descriptions and use the Generalized Search Tree (GiST) algorithm proposed by Hellerstein in [13] for creating and maintaining the repository of numerically encoded services. Combining both encoding and indexing techniques allows performing efficient service search, in the order of milliseconds for trees of 10000 entries. However, insertion within trees of this size is still a heavy process that takes approximately 3 seconds. In [26], the authors propose an approach to optimizing service discovery in a UDDI registry augmented with OWL-S for the description of semantic Web services. In this approach, the authors propose to exploit the service advertisement phase to perform semantic reasoning and pre-compute information that will help to efficiently answer service requests. Performance evaluation of this approach shows that the service publishing phase employing this algorithm takes around seven times the time taken by UDDI to publish a service. On the other hand, the time to process a service request is in the order of milliseconds.

While the two presented approaches opt for overloading the service advertisement phase with costly computations in order to later achieve efficiency upon resolving service requests, we aim at achieving both lightweight service advertisement and lightweight request resolution, as pervasive service discovery needs to be performed as well on resource constrained-devices. Hence, we carry out the resource-demanding semantic reasoning on ontologies offline, i.e., neither upon service advertisement nor upon service request. In our EASY approach to efficient semantic service discovery, described in Section 4, classified ontologies are encoded, and are intended to be used as such by service developers for the semantic annotation of services and service requests, which enables efficient service matching. Furthermore, encoded service descriptions are effectively classified in service repositories or caches, which enables scalable, efficient service insertion and retrieval. In the following section, we first introduce EASY-L and EASY-M, our semantic service description language and associated matching approach.

3 EASY-Language and -Matching: Semantic, Context- and QoS-aware Service Description and Matching in Pervasive Computing Environments

3.1 EASY-Language

Service discovery in pervasive environments calls for a language enabling the semantic, context- and QoS-aware specification of services' advertised capabilities as well as clients' requested capabilities. We introduce EASY-Language (EASY-L), which supports the unambiguous specification of functional and non-functional properties of services and clients. EASY-L is specified using the Web Ontology Language (OWL). The main conceptual elements characterizing EASY-L are depicted in Figure 3. In this diagram, as well as in the two diagrams depicted in Figure 4 and 5, colored boxes repre-

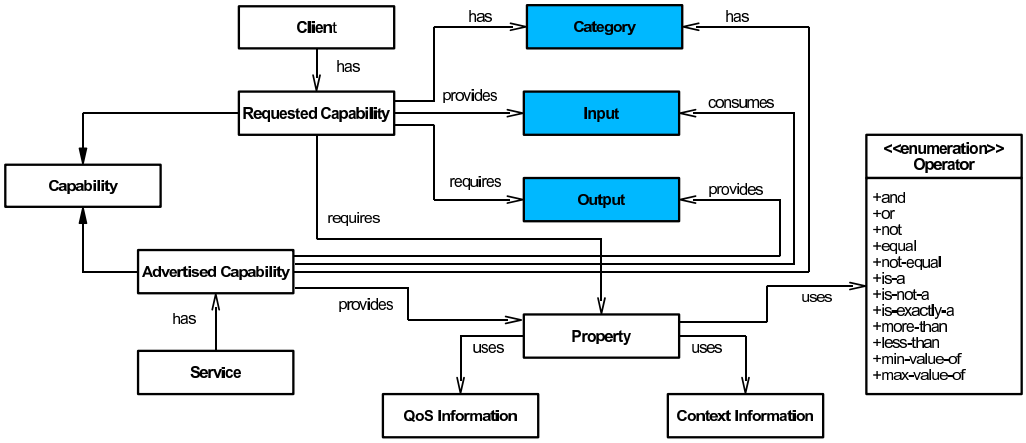


Figure 3: Service Model

sent service information that will be used for service matching (Section 3.2), including functional and non-functional properties of services. These properties can be of two types: qualitative or quantitative. Qualitative properties, represented in the diagrams by dark colored boxes, are those described with reference to ontology concepts (e.g., service inputs, outputs, security levels), whereas quantitative properties, represented in the diagrams by light colored boxes, are those that can be measured and assigned with numeric values (e.g., service latency, service cost, environment temperature).

At the heart of EASY-L (Figure 3), we distinguish the notion of *capability*, which corresponds to the description of any functionality that may be advertised by a service or sought by a client. This description is given in terms of *inputs*, *outputs*, *category* and *properties*. Using our model, a *service* description contains a set of *advertised capabilities*, while a *client* request description contains a set of *requested capabilities*. A requested capability is described with a set of provided inputs and a set of required outputs, a required category and a set of required properties, while an advertised capability is described using a set of required inputs and a set of provided outputs, a provided category and a set of provided properties. Inputs, outputs and category are defined with reference to one or more ontology concepts in a way similar to qualitative non-functional properties defined below.

Non-functional properties (called *property* in the diagram) are expressed in the form of boolean expressions. We actually support the following operators: *and, or, not, equal, not-equal, is-a, is-exactly-a, is-not-a, more-than, less-than, max-value-of, min-value-of.* The operators *is-a, is-exactly-a* and *is-not-a* are used to define qualitative properties, including both functional and non-functional ones, while *equal, not-equal, more-than, less-than, max-value-of, min-value-of* operators are used to define quantitative properties. Finally, the *and, or* and *not* operators are used to define composite

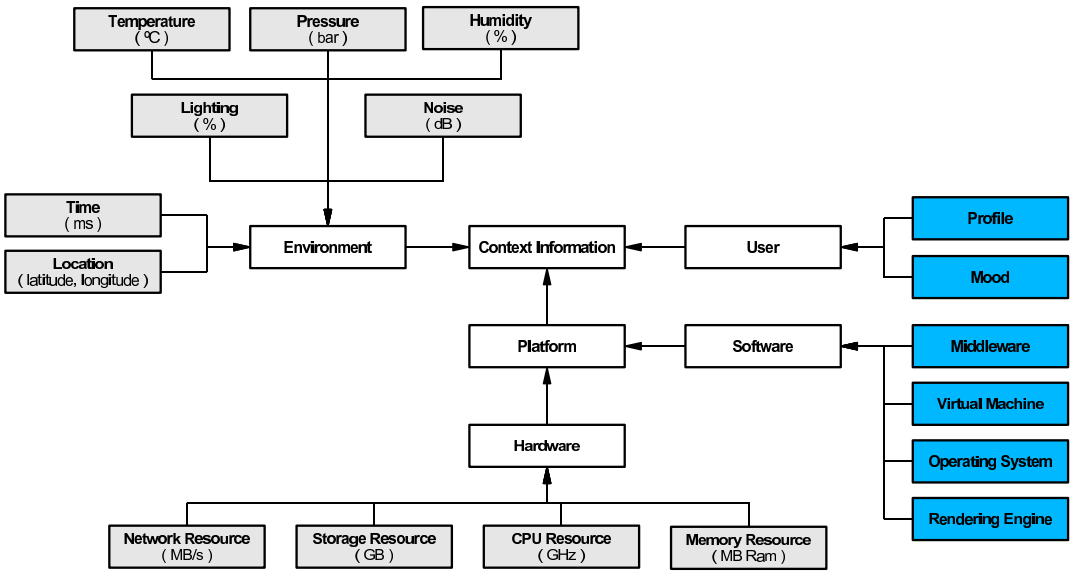


Figure 4: Context Properties

properties. Non-functional properties are related to context and QoS information represented using the context and QoS models depicted in Figure 4 and 5, respectively. These models have been defined based on the context ontology proposed by Preuveneers *et al.* in [21], and the QoS model proposed by Liu and Issarny in [15]. These models contain elementary context and QoS information necessary for a basic specification of non-functional properties of services in pervasive environments, and can be further extended using external ontologies for the definition of additional or more fine-grained service properties. The context model depicted in Figure 4 classifies context information into three main categories: *Environment*, *User* and *Platform*. Environment properties are related to the physical environment, e.g., the environment's temperature. User properties describe information related to the users of the pervasive environment, e.g., user profile and current mood. Finally, platform properties are related to both software and hardware resources of the environment. Software properties describe information related to software applications, the operating system, and the middleware used by these applications, while hardware properties are related to computing and networking resources.

Furthermore, the QoS information related to services is described in the QoS model depicted in Figure 5. In this model, QoS information is divided into five categories (i.e., *Security*, *Transaction*, *Reliability*, *Performance* and *Cost*), while each category decomposes itself into various QoS dimensions. As properties for both QoS and context information may contain expressions with measurement units, default units have been

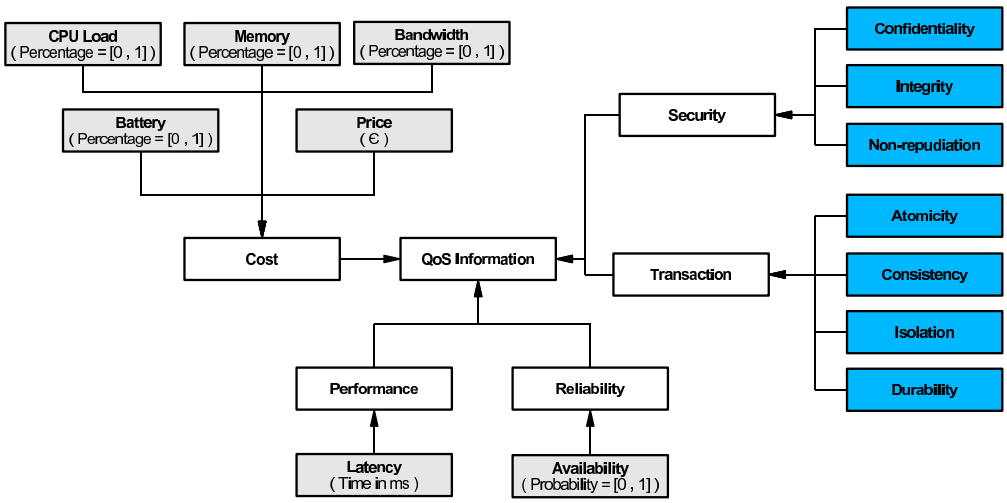


Figure 5: QoS Properties

specified in the model and a separate ontology of units[12] can be further used. This ontology can be used to achieve reusability of unit definitions, and to enable conversion from one unit to another.

An example of a service advertisement and request specification is depicted on the top part of Figure 6. In this example, the requested capability (right part of the figure) is of category *VideoServer*, while the advertised capability (left part of the figure) is of category *DigitalServer*, so it is able to act as a gaming, music or video server, according to the employed ontology depicted in the middle part of the figure[13]. The requested capability has a *Title* as input and the corresponding *VideoResource* as output, while the advertised capability has also a *Title* as input and provides the client with a list of corresponding *DigitalResources*. A number of properties have further been specified regarding the *Network* to be used by the requested capability of the client, which is required to be of type *WirelessNetwork*. Moreover, the requested capability should have a minimum value of *Price* and engender a *Latency* less than 10 units of time, as well as ensure an availability greater than 80%. On the other hand, the advertised capability has a number of properties, including the one related to the employed network connectivity, which is of type *WiFi 802.11g*, the one specifying that using this capability costs 10 units of money, and other QoS related properties such as latency and availability.

3.2 EASY-Matching

Based on EASY-L, defined in the previous section, we now present EASY-Matching (EASY-M) a set of conformance relations for matching services in terms of their func-

[12] Ontology of units: http://www.cs.kuleuven.be/~davy/ontologies/2007/02/Units.owl

[13] A more formal definition of this category would be *Category is-a (VideoServer and MusicServer and GameServer)*

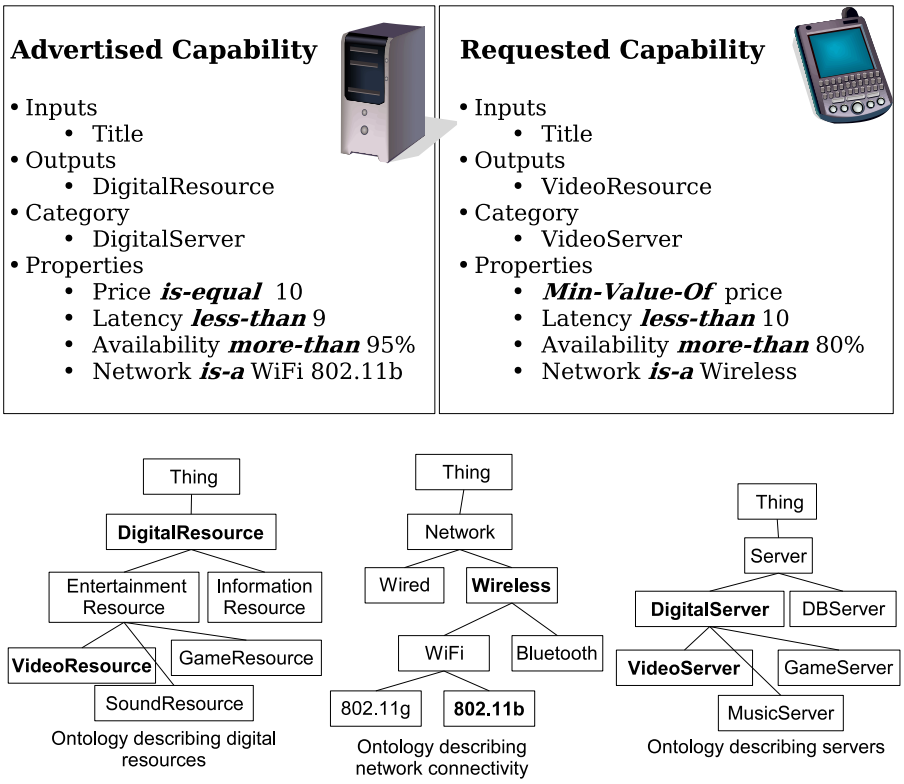


Figure 6: Example of Advertised and Requested capabilities

tional and non-functional properties. First, we introduce three relations for matching the functional properties of a requested capability *Req* and an advertised capability *Adv*, i.e., **ExactCapabilityMatch**(*Adv, Req*), **InclusiveCapabilityMatch**(*Adv, Req*), and **WeakCapabilityMatch**(*Adv, Req*). In these three relations, we use the relation **ConceptMatch**() to compare two concepts c_1, c_2 of an ontology O. This relation is based on the four matching levels defined by Paolucci *et al.* in [18] and discussed in Section 2.2. However, we redefine the exact level of match as holding only if the two compared concepts are the same in O, removing the case where one concept is a direct subclass of the other as part of the plugin level of match.

The **ExactCapabilityMatch**() relation allows to find advertised capabilities that exactly match a requested capability, i.e., only exact matches between concepts are considered. This relation is defined as follows:

ExactCapabilityMatch(*Adv, Req*) =
∀ *in* ∈ *Req*.In, ∃ *in'* ∈ *Adv*.In: **ConceptMatch**(*in', in*) = exact and
∀ *out* ∈ *Req*.Out, ∃ *out'* ∈ *Adv*.Out: **ConceptMatch**(*out', out*) = exact and
ConceptMatch(*Adv.Category, Req.Category*) = exact

The **InclusiveCapabilityMatch**(*Adv, Req*) allows to find capabilities that can as well be more generic than the requested capability, i.e., the inputs, outputs and category of the advertised capability are more generic than the inputs, outputs and category of the requested capability. This relation is defined as follows:

InclusiveCapabilityMatch(*Adv, Req*) =
$\forall$ *in* $\in$ *Req*.In, $\exists$ *in'* $\in$ *Adv*.In: **ConceptMatch**(*in', in*) = exact|plugin and
$\forall$ *out* $\in$ *Req*.Out, $\exists$ *out'* $\in$ *Adv*.Out: **ConceptMatch**(*out', out*) = exact|plugin and
ConceptMatch(*Adv.Category, Req.Category*) = exact|plugin

Finally, the **WeakCapabilityMatch**(*Adv, Req*) relation is the least restrictive matching relation among the three relations: sought concepts of capabilities can either subsume or be subsumed by provided concepts. The consequence of using this relation is that it is possible to find services that provide the client with outputs that are more specific than required (e.g., a car renting capability may only provide particular brands of cars, such as *Peugeot*, rather than providing any *car* as a client may have specified in the service request). Furthermore, the client may provide some inputs that are too generic for the service, leading to a possible malfunction of the latter service (e.g., if the service translates only *Latin* languages into other *Latin* languages, and the client provides as input the concept *Language*, which subsumes both *Greek* and *Latin* languages, the service will not work if the client invokes the service with a text in *Greek* as input). This relation is defined as follows:

WeakCapabilityMatch(*Adv, Req*) =
$\forall$ *in* $\in$ *Req*.In, $\exists$ *in'* $\in$ *Adv*.In: **ConceptMatch**(*in', in*) $\neq$ fail and
$\forall$ *out* $\in$ *Req*.Out, $\exists$ *out'* $\in$ *Adv*.Out: **ConceptMatch**(*out', out*) $\neq$ fail and
ConceptMatch(*Adv.Category, Req.Category*) $\neq$ fail

In the three matching relations the function **ConceptMatch**() can apply to boolean expressions (e.g., a client may specify as output of its required capability: *output1 is-a VideoResource or SoundResource*). In this case, boolean expressions are transformed into the disjunctive normal form and matching is performed for each term of the expression.

From the previous relations note that:
Prop 0: **ExactCapabilityMatch**(*Adv, Req*) $\Rightarrow$ **InclusiveCapabilityMatch**(*Adv, Req*) $\Rightarrow$ **WeakCapabilityMatch**(*Adv, Req*).

If we consider the example of Figure 6, an **InclusiveCapabilityMatch**() relation holds between the advertised and the requested capabilities, as the input, output and category of the advertised capability are all equivalent or more generic than the input, output and category of the requested capability. Specifically, **ConceptMatch**(*Title,Title*)=exact, **ConceptMatch**(*DigitalResource,VideoResource*)=plugin and **ConceptMatch**(*DigitalServer,Video-Server*)=plugin.

Furthermore, when a match holds between a set of advertised capabilities Adv_i and a requested capability *Req*, we use the function **CapabilityDegreeOfMatch**() to rate each advertised capability Adv_i with respect to *Req*. This function is based on the function **ConceptDegreeOf-**

Match() between two concepts c_1 and c_2 in an ontology O. **ConceptDegreeOfMatch**(c_1, c_2) is defined as follows:

ConceptDegreeOfMatch(c_1, c_2)
= 0, if **ConceptMatch**(c_1, c_2) = exact
= number of levels between c_1 and c_2, if **ConceptMatch**(c_1, c_2) = plugin|subsume
= NULL, if **ConceptMatch**(c_1, c_2) = fail

The definition of the function **CapabilityDegreeOfMatch**() is then given by:

CapabilityDegreeOfMatch(*Adv, Req*) =
$w_p(\sum_{i=1}^{n_1}$ **ConceptDegreeOfMatch**(c_i, c'_i) where **ConceptMatch**(c_i, c'_i) = plugin) +
$w_s(\sum_{i=1}^{n_2}$ **ConceptDegreeOfMatch**(c_i, c'_i) where **ConceptMatch**(c_i, c'_i) = subsume)

where n_1 and n_2 are, respectively, the number of $plugin$ and $subsume$ matches recognized between concepts of Adv and concepts of Req. Furthermore, w_p and w_s are weights given to the two levels of match $plugin$ and $subsume$ respectively, such that $w_p \leq w_s$. These weights are used to specify the relative importance of the $plugin$ and $subsume$ levels of match between concepts. If w_p is specified as strictly lower than w_s, then this means that more generic concepts are preferred to specific ones. Note that the $exact$ level of match is not given a weight as the corresponding **ConceptDegreeOfMatch**() is equal to 0. Finally, for the evaluation of the **CapabilityDegreeOfMatch**() function, only pairs of concepts such that the relation **ConceptMatch**() holds, i.e., it is not equal to $fail$, are considered.

In the previous example of Figure 6, the **CapabilityDegreeOfMatch**() between the advertised and the requested capabilities is equal to $3*w_p$, which is calculated as follows: Two $plugin$ matches are identified between the requested and the advertised capabilities, i.e., **ConceptMatch**(*DigitalResource, VideoResource*) = plugin and **ConceptMatch**(*DigitalServer, VideoServer*) = plugin. As such, using the ontologies depicted in the middle of Figure 6, we have:

ConceptDegreeOfMatch(*DigitalResource, VideoResource*) = 2 and
ConceptDegreeOfMatch(*DigitalServer, VideoServer*) = 1.

Hence, **CapabilityDegreeOfMatch**(*Adv, Req*) = $w_p * (2 + 1)$.

After rating the advertised capabilities *Adv*$_i$, the ones such that **CapabilityDegreeOfMatch**(*Adv*$_i$, *Req*) is the lowest are preferred. Nevertheless, the fulfillment of non-functional properties is also checked. Specifically, all the required properties specified in the requested capability description are compared with provided properties of the advertised capabilities. Quantitative properties (e.g., latency *less-than* 5, temperature *more-than* 20, price *less-than* 50) are numerically compared, whereas qualitative properties (e.g., NetworkConnectivity *is-exactly-a* BluetoothConnectivity, VirtualMachine *is-a* JVM5) are checked using the relation **ConceptMatch**().

If two or more advertised capabilities have the same **CapabilityDegreeOfMatch**() and meet all the required properties of the requested capability, we select the capability that best matches the required properties of the requested capability. This is done using the **PropertiesDegreeOfMatch**() function defined as follows:

$$\textbf{PropertiesDegreeOfMatch}(\textit{Adv, Req}) = \sum_{i=1}^{n_i} w_i * p_i \quad (1)$$

where w_i is the relative importance of the considered property, i.e., the higher the weight w_i assigned to the property p_i is, compared to the weights assigned to the other properties, the more

p_i is preferred in relation to other properties. This allows a client to specify priorities between non-functional properties. For instance, a client may prefer using a service that ensures a higher security level even if this service has lower latency than other services. In this case, the weight given to the property $Security$ should be greater than the weight given to the property $Latency$.

In nature, finding a good match is reduced to a problem where multiple (possibly contradicting) objectives need to be satisfied. The use of weights assigned to the properties reduces the problem to a single-objective optimization problem. A different approach would reduce the problem to a Multi-objective Optimization Problem (MOP). Rather than finding a single solution as in global optimization, the goal would be to find good compromises or trade-offs, resulting in a set of solutions often referred to as the *Pareto optimal set*. The QoS properties corresponding to solutions in the optimal set are called *non-dominated*, i.e. for a given set of constraints, some QoS properties may affect the outcome of the optimization problem less than those in the optimal set. As the relative importance of the QoS requirements may differ from one another, further research will have to investigate whether solving the optimization problem to determine the *Pareto optimal set*, graphically represented as a *Pareto front*, would be feasible and lead to other results.

Since properties are heterogeneous − i.e., some are qualitative, some are quantitative and further expressed in different units − data normalization is needed in order to evaluate the **PropertiesDegreeOfMatch**(). The first normalization that we introduce is assigning numeric values to qualitative properties such that they can participate in the **PropertiesDegreeOfMatch**() function. These values are given by the function **ConceptDegreeOfMatch**() defined earlier. This allows evaluating a provided qualitative property with respect to a required property. Indeed, the smaller the **ConceptDegreeOfMatch**() between a provided qualitative property and a required one is, the better. The second normalization that we apply is the standard deviation normalization on the various properties as in [15]. This normalization is as follows:

Properties that are stronger with larger values (e.g., availability) are normalized according to the following equation:

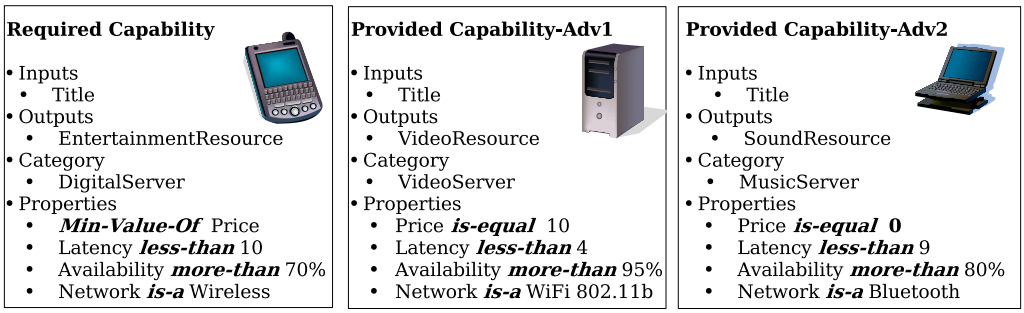
$$p'(adv_i) = \begin{cases} 1 & \text{if } (p(adv_i) - m(p) > 2 * \delta(p)) \\ 0 & \text{if } (p(adv_i) - m(p) < -2 * \delta(p)) \\ \frac{p(adv_i) - m(p)}{4 * \delta(p)} + 0.5 & \text{otherwise} \end{cases} \quad (2)$$

While properties that are stronger with smaller values (e.g., latency, normalized qualitative properties), are normalized according to the following equation (so that smaller values contribute more to the **PropertiesDegreeOfMatch**() function):

$$p'(adv_i) = \begin{cases} 0 & \text{if } (p(adv_i) - m(p) > 2 * \delta(p)) \\ 1 & \text{if } (p(adv_i) - m(p) < -2 * \delta(p)) \\ 0.5 - \frac{p(adv_i) - m(p)}{4 * \delta(p)} & \text{otherwise} \end{cases} \quad (3)$$

where $p(adv_i)$ is the value of property p for the advertised capability adv_i, and $m(p)$ and $\delta(p)$ are the mean value and standard deviation for the property p, respectively.

An example of evaluating the **PropertiesDegreeOfMatch**() is depicted in Figure 7. In this example, two advertised capabilities (*Adv*$_1$ and *Adv*$_2$) are matched with the requested capability *Req*. The requested capability is specified on the device of a traveling user who is looking for entertainment capabilities in the various pervasive environments that he crosses during his travel. In particular, this user is looking for digital servers, which he can access, giving the title of an entertainment resource and receiving in result the corresponding resource. Currently, the user is waiting for his train in a big city train station. The requested capability on the user device further identifies some required properties regarding latency and availability, as well as network connec-



	Req	Adv1	Adv2	Adv1'	Adv2'	Adv'1	Adv'2
In	Title	Title	Title	0	0	CDM= $2w_s$	CDM= $2w_s$
Out	Entertainment	VideoResource	SoundResource	1	1		
Category	Digital Server	VideoServer	Music Server	1	1		
Price	Min	10	0	10	0	0,32	0,68
Latency	<10	<4	<9	4	10	0,68	0,32
Availability	>70%	>95%	>80%	95,00%	80,00%	0,68	0,32
Network	WirelessConnection	WiFi802.11b	Bluetooth	1	2	0,68	0,32

PropertiesDegreeOfMatch(*Adv1,Req*)=0,32*w_1+0,68*w_2+0,68*w_3+0,68*w_4=**2,36** (if w_i=1)
PropertiesDegreeOfMatch(*Adv2,Req*)=0,68*w_1+0,32*w_2+0,32*w_3+0,32*w_4=**1,64** (if w_i=1)

Figure 7: Matching non-functional Properties Example

tivity and price. In this example, two entertainment capabilities are available in the environment. Specifically, a *VideoServer* and a *MusicServer*. The video server is offered by the networking infrastructure of the train station, thus providing strong QoS properties (e.g., high availability, low latency). However this capability is not free-of-charge. On the other hand, another traveler in the train station allows other users to use his music resources for free, but without good QoS guaranties. Both advertised capabilities match the requested capability of Figure 6. These two advertised capabilities have the same value of **CapabilityDegreeOfMatch**() (noted CDM in the figure) with the requested capability, i.e., equal to $2w_s$. Hence, for selecting the best among the two advertised capabilities, we use the **PropertiesDegreeOfMatch**() function. Towards this purpose, we first normalize qualitative properties to numeric values using their **ConceptDegree-OfMatch**() (results are given in columns *Adv1'* and *Adv2'* of the table for the advertisements *Adv1* and *Adv2* respectively). Using these values we then normalize all the properties using the standard deviation normalization (results are given in columns *Adv1"* and *Adv2"*). Having all the values normalized, it is easy to evaluate the **PropertiesDegreeOfMatch**() for each advertised capability as follows:

PropertiesDegreeOfMatch(*Adv1, Req*) = $0.32 * w_1 + 0.68 * w_2 + 0.68 * w_3 + 0.68 * w_4$,
PropertiesDegreeOfMatch(*Adv2, Req*) = $0.68 * w_1 + 0.32 * w_2 + 0.32 * w_3 + 0.32 * w_4$

where w_1, w_2, w_3 and w_4 are the weights of each of the properties $Price$, $Latency$, $Availability$ and $Network$, respectively. Assuming that they all have the same relative importance, i.e., $w_i = 1$, EASY-M will select the advertised capability $Adv1$, as it presents the highest degree of match for properties, i.e., $2.36 > 1.64$.

In this section, we presented EASY-L, a language for semantic, context- and QoS-aware service specification, as well as EASY-M, a solution for matching requested and advertised capabilities in pervasive computing environments. Nevertheless, EASY-M, and in particular the underlying **ConceptMatch**() relation, relies on the costly reasoning on ontologies used to assess relations between concepts. In the following two sections, we present solutions for efficient service discovery of service capabilities.

4 EASY: Efficient Semantic Service Discovery in Pervasive Computing Environments

In this section, we present our solution for efficient semantic discovery of service capabilities. This solution decomposes into two parts. First, we focus on the optimization of semantic reasoning on ontologies in order to efficiently infer relations between ontology concepts. Our approach, described in Section 4.1, allows to quickly find whether two concepts are related to each other in an ontology or not, without performing the costly semantic reasoning online. Specifically, we propose to perform classification of ontologies, which involves semantic reasoning, offline, and to encode the classified ontology hierarchies. Concepts involved in service and request descriptions are then annotated with their corresponding codes in the encoded hierarchies, which reduces the semantic matching of concepts performed by the **ConceptMatch**() function to a numeric comparison of codes. In the second part of our solution, described in Sections 4.2 and 4.3, we introduce an overview of the architecture of EASY that enables the transparent deployment of semantic service discovery on top of existing SDPs. We further present our mechanisms for organizing services according to their semantic similarity within a service repository. This allows efficiently advertising services and resolving service requests.

4.1 Encoding Semantic Concept Hierarchies

As discussed in Section 2.4.1, encoding object class hierarchies with multiple inheritance for quickly performing subtype testing has been a very active field of research in the last decade. However, there are several reasons why the previously discussed subtype encoding techniques are not so useful for representing subsumption within ontologies. First of all, incremental encoding for ontologies assumes an *open world* view on the concepts involved, whereas a compiler knows all types in a program (*closed world* assumption) and can do a more efficient and compact encoding. The compiler can assume that no other classes are being subsumed if such classes do not exist, which is not the case with ontologies. Another reason is that for ontologies we require scalability to very large ontologies with support for conflict-free incremental encoding so as to easily reuse previously encoded concepts. Some encoding techniques provide support for incremental encoding, but require the re-encoding of conflicting codes whenever a subsumption test results in a false positive. For ontologies, where reuse of concepts defined therein is one of the main goals, the encoding should be carried out once offline and be reused as much as possible.

To deal with the above requirements, we have developed a prime-based encoding technique for subsumption testing of classes in ontologies that yields a new way of compaction, supporting incremental encoding by avoiding conflicts rather than solving them. The algorithm exploits the sparseness of the binary matrix representation by only encoding a reference to ancestors of a class; this is done by assigning a unique prime number π_i as a personal gene $g_i \in G$ to each class $C_i \in \chi$ in the hierarchy χ. Unique prime numbers ensure the conflict-free incremental encoding. The encoding of a multiple inheritance hierarchy $\chi = \{A, B, C, D, E\}$ is illustrated in Figure 8. Class E inherits the genes 2 and 7 from its ancestors and is assigned 11 as its

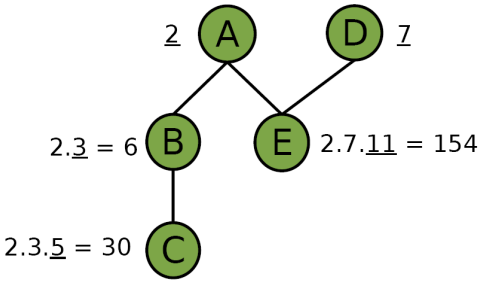


Figure 8: Hierarchy encoding using prime numbers

personal gene g_E. Personal genes are underlined in Figure 8. Encoding of a class C_i is given by the relation $\gamma(C_i) = \prod_j g_j$, with g_j the genes of class C_i and its ancestors. For instance, for the class E, this relation yields: $\gamma(E) = 2 * 7 * 11 = 154$. As classes A and D have no ancestors, their encoding $\gamma(A)$ and $\gamma(D)$ is solely based on their personal gene $g_A = 2$ and $g_D = 7$.

A class $A \in \chi$ is subsumed by a class $B \in \chi$ if the gene $g_B = \varphi(B)$ of class B divides the encoding $\gamma(A)$ of class A. For incrementally encoding a new class C_{n+1}, the next available prime number $\pi_{n+1} = \varphi(C_{n+1})$ is used as a personal gene. $\gamma(C_{n+1})$ can be computed without traversing the hierarchy to collect the genes of the ancestors, but by using the *least common multiple* function $lcm(\{\ \gamma(C_a), \gamma(C_b), \gamma(C_c), ...\ \})$ of the encoding of its parents $C_a, C_b, C_c, ...$ Note that our incremental encoding only ensures conflict-free encoding, but not the most compact representation.

4.1.1 Optimizing the Encoding Length of the Representation

The order of assigning prime numbers in Figure 8 was randomly chosen. This may not only affect the encoding length of the class itself, but also that of its descendants. By selecting a prime number for each class, more compact representations can be achieved. In our algorithm, we assign prime numbers π_i in order, (i.e., 2, 3, 5, ...) and use the following heuristics:

- *Minimize total encoding length*: Ancestors with the most descendants are encoded first, hence using the smallest prime numbers, to achieve the shortest total encoding length of the hierarchy for all classes together.
- *Minimize longest encoding length*: To minimize the longest encoding of a single class in the hierarchy, the least possible encoding of the leaves of the hierarchy is estimated given the current incremental encoding.

Our first algorithm illustrates the use of the first heuristic to encode the hierarchy while ensuring the most compact representation of the hierarchy for all classes together. The algorithm iteratively selects the next best class and assigns it a prime number until all the classes have been processed. For a multiple inheritance hierarchy containing n classes $C_1, C_2, ..., C_n$, (Algorithm 1) shows the encoding of the hierarchy to achieve the set of codes for each class $\gamma(C_1)$, $\gamma(C_2)$, ..., $\gamma(C_n)$. On the other hand, if minimizing the longest encoding of a class $\gamma(C_i)$ is required, then the selection of *bestClass* is changed by investigating what the largest encoding would be if all descendants of class C_i would immediately receive a next prime. The one with the largest predicted leaf is chosen as the *bestClass* in that case. After the gene assignment has been carried out, each of the n classes C_i has a personal gene and a code $\gamma(C_i)$ computed by

Algorithm 1 EncodeHierarchy(in: hierarchy, **out:** gamma[])

```
 1: n = SizeOf(hierarchy)
 2: primeTable[1..n] = ComputePrimes(n)
 3: i = 0
 4: classList = Roots(hierarchy)
 5: while SizeOf(classList) > 0 do
 6:     bestClass = First(classList)
 7:     for each C in classList do
 8:         if SizeOf(Descendants(C)) > SizeOf(Descendants(bestClass))) then
 9:             bestClass = C
10:         end if
11:     end for
12:     i = i + 1
13:     AssignPersonalGene(bestClass, primeTable[i])
14:     AddInheritedGene(Descendants(bestClass), primeTable[i])
15:     RemoveClass(classList, bestClass)
16:     AddClass(classList, NoPersonalGene(Children(bestClass)))
17: end while
18: for each C in hierarchy do
19:     gamma[C] = MultiplyAllGenes(C)
20: end for
```

multiplying the personal gene with the inherited genes. The following heuristics can then be used to test for subsumption:

- A class $A \in \chi$ never inherits from another class $B \in \chi$ if the encoding $\gamma(A)$ of class A is smaller than the encoding $\gamma(B)$ of class B.
- A class $A \in \chi$ never inherits from another class $B \in \chi$ if the personal gene $\pi(A)$ of class A is smaller than the gene $\pi(B)$ of class B.

For two classes C_1, C_2 in a hierarchy that have encoding $\gamma(C_1), \gamma(C_2)$ and genes π_1, π_2, the subsumption algorithm for testing if one class C_1 subsumes C_2 is given in (Algorithm 2). In

Algorithm 2 Subsumes(in: C_1, C_2**)**

```
1: if π1 > π2 then
2:     return false
3: end if
4: if γ(C1) > γ(C2) then
5:     return false
6: end if
7: return (γ(C2) % π1 = 0)
```

our implementation, we have also developed an optimized version of the modulo operator that avoids any logic for determining the correct sign of the result. In fact, we even eliminate the computing of the quotient or the remainder, but just test whether the remainder is zero or not. A performance evaluation of our encoding mechanism is presented in Section 5.2.

Under the assumption that the classified ontologies are encoded and that service advertisements and service requests have been annotated with the codes corresponding to the concepts that they involve, semantic service reasoning reduces to a numeric comparison of codes. Indeed, to infer whether a concept C_1 represented by the code α subsumes another concept C_2 represented by the code β, it is sufficient to numerically compare α with β, i.e., check whether the gene of α divides β, which is an elementary processor operation that has a cost in the order of nanoseconds. In order to ensure consistency of codes in the face of the dynamics and

evolution of ontologies, service advertisements and service requests specify the version of the codes being used. Specifically, each involved concept is annotated with the triple: <*Ontology, Code, Version*>, where *Ontology* is the URI of the ontology describing the concept, *Code* is the code given to the concept by the encoding algorithm, and *Version* is the version of the code, which evolves along with the ontology evolution. We assume that services periodically check the version of codes that they are using and update their codes in the case of ontology evolution.

Performing efficient matching of service capabilities constitutes a first significant optimization towards enabling the deployment of semantic solutions in pervasive environments: thin devices can efficiently carry out semantic matching without the need to host and run highly resource-consuming reasoners. Nevertheless, with the increasing number of services in the environment, solutions for reducing the number of semantic matchings performed to resolve a service request are required. Our second related optimization relies on indexing and classifying service advertisements according to their semantic similarity. Before introducing this optimization in Section 4.3, we first give in the following section, an overview of the overall EASY architecture.

4.2 EASY Architecture Overview

As previously discussed in the introduction, EASY is not yet-another SDP. Instead, EASY allows enhancing existing SDPs with support of semantic-based matching. Figure 9 shows an overview of the architecture of EASY, which enables the transparent deployment on top of existing SDPs. In this figure, a legacy SDP is depicted with its provided mechanisms for service matching, called SDP-M, and its provided mechanisms for storing service descriptions given in SDP-L[14]. Deploying EASY on top of this SDP enables dealing with both the EASY-L semantic descriptions and the SDP-L syntactic descriptions. Indeed, semantic service advertisements/requests are managed by the introduced EASY mechanisms while the syntactic service advertisements/requests are managed by the initial SDP stack. An index that links the semantic EASY-L descriptions with their corresponding SDP-L descriptions is maintained. The introduced EASY mechanisms further discussed in the following section include: EASY-M for matching EASY-L service descriptions and a solution for grouping EASY-L service descriptions towards efficient service insertion and retrieval. In the case of a centralized legacy SDP, the deployment scheme shown in this figure applies only at the centralized registry. In the case of a fully distributed SDP, where caches are maintained for temporary storing service advertisements, the deployment of EASY should be performed in all the nodes of the network that participate in the discovery process. An example of the deployment of EASY on top of a semi-distributed SDP in discussed in Section 5.

4.3 Organizing Services in the Repository

Based on the EASY-M conformance relations of Section 3.2 and according to the architecture described in the previous section, we present in this section a solution for semantically organizing a service repository in order to minimize the number of matchings performed to answer a service request (Section 4.3.2). Our solution further enables efficient insertion of a service advertisement in the repository, minimizing the number of matchings performed for this purpose with advertisements already registered (Section 4.3.1).

Finally, each performed matching is itself carried out efficiently without involving any semantic reasoning thanks to the encoding technique presented in Section 4.1.

[14]SDP-L and SDP-M are respectively the legacy service description language and matching mechanisms related to the legacy SDP

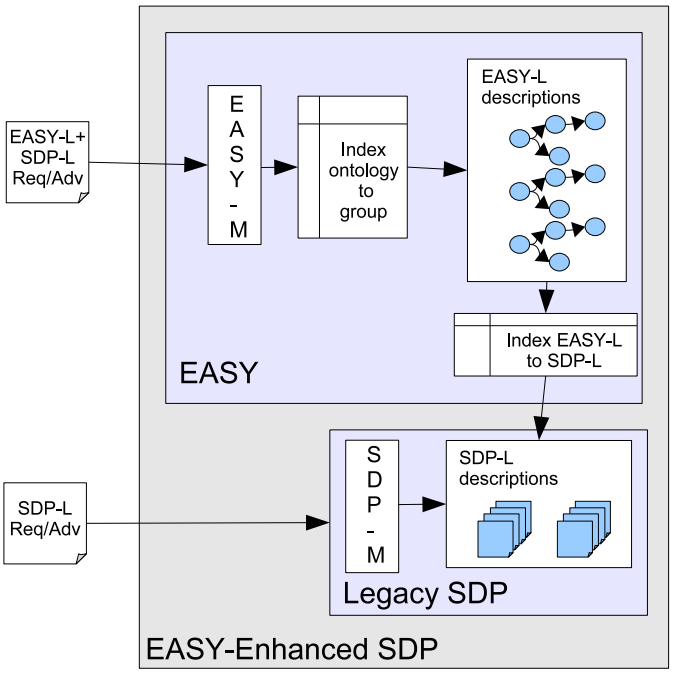


Figure 9: EASY Architecture Overview

4.3.1 Adding a New Service Advertisement

At a pre-processing phase, our approach constructs directed acyclic graphs (DAGs) of capabilities of the advertised services. These graphs are indexed according to the ontologies being used in the capabilities that they contain as shown in Figure 9. More precisely, an index table is maintained for specifying which ontologies are used for each graph. To construct the graph we employ the relations **ExactCapabilityMatch**() and **InclusiveCapabilityMatch**() defined in Section 3.2. Specifically, if **ExactCapabilityMatch**(C_1, C_2) holds between two service capabilities C_1 and C_2, then these capabilities will be represented by a single node in the graph. On the other hand, if **InclusiveCapabilityMatch**(C_1, C_2) holds and **ExactCapabilityMatch**(C_1, C_2) does not, C_1 and C_2 will be represented by two distinct nodes with a directed edge from C_1 to C_2.

Using this grouping technique, the most *generic* capabilities will be represented by root nodes in graphs G_i, noted **Roots**(G_i), i.e., nodes of G_i that do not have predecessors in G_i. These capabilities are said to be more generic than other capabilities contained in their subhierarchy because they provide outputs, inputs and category that subsume the respected ones of their successors in the graph. For instance, the advertised capability depicted in Figure 6, which provides digital resources is *more generic* than the two advertised capabilities depicted in Figure 7, which are providing only specific types of digital resources, i.e., Video and Music, respectively. Similarly, we define **Leaves**(*G*) as the set of nodes in the graph *G* that do not have successors in *G*. The capabilities classified in the set **Leaves**(*G*) of a graph *G* are the most *specific* capabilities of the graph, which means that they provide outputs, inputs and category that are subsumed by the respected ones of their predecessors in the graph.

Algorithm 3 InsertService(in: serviceDescription, $G_{1..m}$, **out:** $G'_{1..k}$)

```
1: for each C_i in serviceDescription do
2:   for each G_i using the same ontologies as C_i do
3:     while not C_i inserted do
4:       for each Root_i in Roots(G_i) do
5:         if not InclusiveCapabilityMatch(Root_i, C_i) then
6:           for each Leaf_i in Leaves(G_i) do
7:             if InclusiveCapabilityMatch(C_i, Leaf_i) then
8:               Test with N_i ∈ predecessors of Leaf_i
9:               until ¬ InclusiveCapabilityMatch(C_i, Pred(N_i))
10:              Draw an edge from C_i to N_i
11:            end if
12:          end for
13:        else
14:          Test with N_i ∈ successors of Root_i
15:          until ¬ InclusiveCapabilityMatch(Succ(N_i), C_i)
16:          Draw an edge from C_i to N_i
17:          for each Leaf_i in Leaves(G_i) do
18:            if InclusiveCapabilityMatch(C_i, Leaf_i) then
19:              Test with N_i ∈ predecessors of Leaf_i
20:              until ¬ InclusiveCapabilityMatch(C_i, Pred(N_i))
21:              Draw an edge from C_i to N_i
22:            end if
23:          end for
24:        end if
25:      end for
26:    end while
27:  end for
28: end for
```

When a new service is registered with a repository, the set of capabilities that it provides are classified among the existing graphs. The algorithm of classifying new capabilities in the existing graphs is described later in this section. This algorithm is based on the following two properties:

Prop 1 : ¬ **InclusiveCapabilityMatch**(*Root*$_i$, *Adv*): *Root*$_i \in$ **Roots**(*G*) $\Rightarrow$
$\forall$ *C* $\in$ **Successors**(*Root*$_i$): ¬ **InclusiveCapabilityMatch**(*C*, *Adv*)

Prop 2 : ¬ **InclusiveCapabilityMatch**(*Adv*, *Leaf*$_i$): *Leaf*$_i \in$ **Leaves**(*G*) $\Rightarrow$
$\forall$ *C* $\in$ **Predecessors**(*Leaf*$_i$): ¬ **InclusiveCapabilityMatch**(*Adv*, *C*)

The proofs of these properties, as well as of the property (Prop 3) introduced in the next section, are given in Appendix A. The properties (Prop 1) and (Prop 2) are used to check whether an advertised capability *Adv* will have, respectively, a predecessor and/or a successor in the graph *G*, without applying the **InclusiveCapabilityMatch**() with all the nodes of a graph. Indeed, if the **InclusiveCapabilityMatch**() fails between a node *Root*$_i$ in **Roots**(*G*) and *Adv*, it will also fail with all the successors of *Root*$_i$ in *G*, i.e., *Adv* will not have a predecessor in *G*. Respectively, if the **InclusiveCapabilityMatch**() fails between *Adv* and a node in **Leaves**(*G*), it will also fail with all the predecessors of *Leaf*$_i$, i.e., *Adv* will not have a successor in *G*. The steps for classifying the capabilities of a new service within a set of graphs G_1, G_2,..., G_n is given in (Algorithm 3).

For each capability C_i advertised by the new service, the algorithm tries to find a graph G_i in which this capability will be integrated (lines 1..3). A subset of graphs is preselected according to the ontologies being used by C_i. The algorithm first checks whether C_i can be inserted in the sub-hierarchy of one of the root nodes of *G*. This is done by verifying if there exists a

node $Root_i$ in **Roots**(G_i) such that **InclusiveCapabilityMatch**($Root_i$, C_i) holds (line 5). Note that, for the sake of simplicity, we sometimes use nodes instead of capabilities as parameters of matching relations (e.g., the use of $Root_i$ in **InclusiveCapabilityMatch**($Root_i$, C_i)). For such notations, we mean that the matching is performed with one of the capabilities represented by the corresponding node, instead of the node itself, as all the capabilities represented by a node are semantically equivalent.

If **InclusiveCapabilityMatch**($Root_i$, C_i) holds (line 13), then C_i will have a predecessor in G_i. The next step is to find this node, N_i, among the successors of the node $Root_i$, such that **InclusiveCapabilityMatch**($Succ(N_i)$, C_i) fails, and to draw an edge from C_i to N_i. Moreover, C_i could have a successor in G_i. Thus, the algorithm tries to find among the set **Leaves**(G_i) if there is a node $Leaf_i$ such that **InclusiveCapabilityMatch**(C_i, $Leaf_i$) holds (line 17)). If **InclusiveCapabilityMatch**(C_i, $Leaf_i$) holds, then C_i will have a successor in G_i. The next step is to find this node, N_i, among the predecessors of $Leaf_i$ such that **InclusiveCapabilityMatch**(C_i, $Pred(N_i)$) fails, and to draw an edge from C_i to N_i (line 21)). On the other hand, if **InclusiveCapabilityMatch**($Root_i$, C_i) does not hold (line 5), C_i will not have a predecessor in G_i. Nevertheless, C_i could have a successor in G_i. Thus, the algorithms checks whether there is a node $Leaf_i$ in **Leaves**(G_i) such that **InclusiveCapabilityMatch**(C_i, $Leaf_i$) holds (lines 6..10). These steps are similar to the aforementioned lines 17..21.

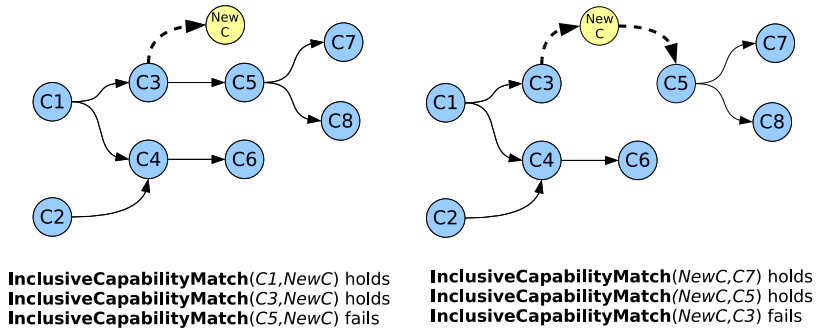


Figure 10: Example of inserting a capability in a DAG

Figure 10 shows an example of inserting a capability, *newC*, in a DAG of capabilities, *G*. The first step (left part of the figure) is to match *newC* with capabilities from **Roots**(*G*) in order to find out whether *newC* will have a predecessor in *G*. Indeed, **InclusiveCapabilityMatch**(C_1, *newC*) holds, which means that one of the successors of C_1 will be linked with *newC*, i.e., C_3. The next step (right part of the figure) is then to find out whether *newC* will have a successor in *G*. This is done by matching the capabilities in **Leaves**(*G*) with *newC*. Indeed, **InclusiveCapabilityMatch**(*newC*, C_7) holds, which means that *newC* will be linked with one of the predecessors of C_7, i.e., C_5.

4.3.2 Resolving Service Requests

When a service request *Req* arrives, our approach first pre-selects, among the existing DAGs, graphs that contain services that are most likely to match the request. This is done using the indexes given to each graph, which correspond to the set of ontologies used by the capabilities of that graph. For each pre-selected graph *G*, the approach performs matching between the request and the capabilities of **Roots**(*G*). Specifically, if **WeakCapabilityMatch**() does not hold

Algorithm 4 MatchService(in: serviceDescription, $G_{1..m}$, **(out:** capabilitySet)

```
1: for each C_i in serviceDescription do
2:    for each G_i using the same ontologies as C_i do
3:       while not C_i matched do
4:          for each Root_k in Roots(G_i) do
5:             if WeakCapabilityMatch(Root_i, C_i) then
6:                Add M to capabilitySet with M in Succ(Root_i)
7:             end if
8:          end for
9:       end while
10:   end for
11:   Select from capabilitySet the capability M such that:
12:   CapabilityDegreeOfMatch(M, C_i) is minimal and
13:   PropertiesDegreeOfMatch(M, C_i) is maximal
14: end for
```

between *Req* and any of the capabilities of **Roots**(*G*), the graph *G* is filtered out, and the next pre-selected graph is checked. This filtering is based on property (Prop 3):

Prop 3 : $\forall$ *Root*$_i \in$ **Roots**(*G*): $\neg$ **WeakCapabilityMatch**(*Root*$_i$, *Req*) $\Rightarrow$
$\forall$ *C* $\in$ G: $\neg$ **WeakCapabilityMatch**(*C*, *Req*)

Prop 4 : $\neg$ **WeakCapabilityMatch**() $\Rightarrow$ $\neg$ **InclusiveCapabilityMatch**() $\Rightarrow$
$\neg$ **ExactCapabilityMatch**()

Note that we use the **WeakCapabilityMatch**() relation instead of the other two relations, i.e., **ExactCapabilityMatch**() and **InclusiveCapabilityMatch**(), to filter out graphs, because it includes the other two relations, as described in property (Prop 4) derived from (Prop 0). On the other hand, if the matching between the request and a capability *Root*$_i$ of **Roots**(*G*) holds, i.e., **WeakCapabilityMatch**(*Root*$_i$, *Req*) holds, the algorithm tries to find a node *C* from the successors of *Root*$_i$, including *Root*$_i$ itself, such that: **CapabilityDegreeOfMatch**(*C*, *Req*) = **Min**(**CapabilityDegreeOfMatch**(C_i, *Req*)), where C_i is a successor of *Root*$_i$ or *Root*$_i$ itself, and C_i meets all the required properties of *Req*.
Note that for finding a capability that exactly matches or is more generic than the requested capability *Req*, excluding capabilities that use more specific concepts than *Req*, an additional test, i.e., **ExactCapabilityMatch**(C_i, *Req*) or **InclusiveCapabilityMatch**(C_i, *Req*), should be performed with the selected node C_i, respectively.

Finally, selection among capabilities represented by the selected node is based on the **PropertiesDegreeOfMatch**() relation. Indeed, as capabilities represented by the same node of a graph are semantically equivalent, we select the capability that best matches the request in terms of non-functional properties.

Given a set of capabilities $C_1, C_2, ..., C_n$ requested in the service description and a set of graphs $G_1, G_2, ..., G_m$, (Algorithm 4) resolves the service requests and returns a set of capabilities of advertised services that best match the input capabilities.

An example concerning the first steps of our algorithm for matching a requested capability with advertised capabilities of services is given in Figure 11. In this figure, the requested capability *Req* uses the ontology O_1 in its specification. This allows to filter out DAG_2, as it is indexed with only the ontology O_3. The next step is to match *Req* with capabilities from **Roots**(DAG_1) and **Roots**(DAG_3), i.e., the capabilities C_1 and C_4. If the matching fails with one of these capabilities, we can infer that no capability will match *Req* in the corresponding graph.

The benefits of using the introduced solution is that it minimizes the number of semantic matchings performed to answer a query. Indeed, thanks to the indexing of graphs, matching is

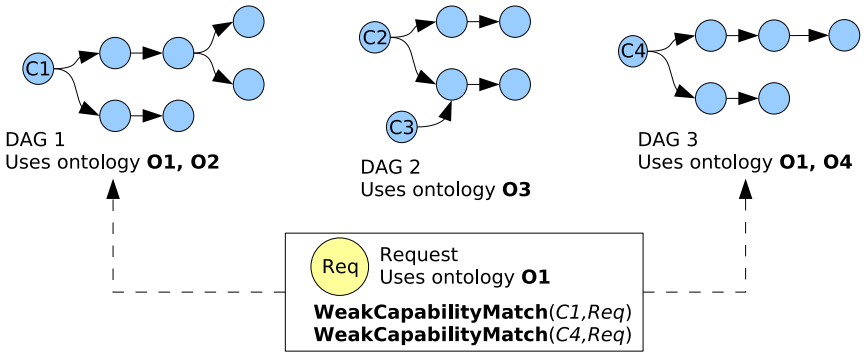


Figure 11: Example of matching a user's requested capability

only performed with graphs that use the same ontologies as the query. Furthermore, graphs that will not provide capabilities that match the query are quickly filtered out by matching the query only with root nodes of those graphs.

As the main function of both service insertion and matching algorithms is parsing a set DAGs and performing matches on the capabilities of the visited nodes its complexity can be approximated with the complexity of elementary graph algorithms (e.g., the breadth-first search algorithm whose complexity is linear in the size of the graph). Furthermore, the processing done in each node for matching capabilities is composed of a set of divisions for assessing subsumption between concepts and is thus linear in the number of concepts being compared (i.e., inputs, outputs, category of advertised and requested capabilities).

Our overall EASY solution to efficient semantic service discovery, combining the substitution of semantic reasoning and the minimization on the number of matching iterations within a repository, complements the EASY-L and EASY-M instruments for describing and matching services. In the next section we evaluate our overall EASY solution.

5 Prototype Implementation and Performance Evaluation

5.1 EASY-Ariadne: Deployment on Top of Ariadne

EASY is intended to be deployed on top of existing SDPs in order to provide them with support for efficient semantic, context- and QoS-aware, service discovery, which are essential requirements for service discovery in pervasive environments. We have deployed EASY on top of Ariadne[15], a scalable semi-distributed service discovery protocol for MANETs [22] according to the architecture presented in Section 4.2. Ariadne relies on a backbone of repositories storing WSDL descriptions and constituting a *virtual network*. Repositories are dynamically deployed, each repository performing service discovery in its vicinity. As such, service discovery in the global network is based on collaboration among deployed repositories. Ariadne can be configured as a fully distributed SDP, if all the networked nodes act as repositories, as well as a centralized or semi-distributed SDP if only one, or a subset of networked nodes, respectively, act

[15] Ariadne SDP: http://www-rocq.inria.fr/arles/download/ariadne/

as repositories. This allows assessing the flexibility of EASY to be employed on top of various SDPs with different deployment schemes.

EASY-Ariadne, i.e., Ariadne augmented with EASY, decomposes into a local and a global discovery process. The local discovery process is performed by each repository. Each repository is thus responsible for:

(i) storing the EASY-L descriptions of the services available in its vicinity, and organizing the capabilities provided by these services according to the DAG-based scheme discussed in Section 4, and

(ii) periodically advertising its presence to its vicinity.

When a repository receives a service request, specified using EASY-L, it seeks capabilities of registered services that semantically match the requested service, as discussed in Section 4.

To deal with the dynamics of pervasive networks, repositories are dynamically and homogeneously deployed in the network using an on-the-fly election process. Specifically, if for a given period of time, a node does not receive any repository advertisement, the node initiates the election of a new repository. The election process is done by broadcasting an election message in the network up to a given number of hops. Thereafter, nodes can either accept or refuse to act as a repository, depending on a number of parameters such as network coverage, mobility and remaining/available resources. This mechanism allows electing repositories with the best properties, and distributing them efficiently, as an election process is launched in the insufficiently covered areas.

The global service discovery process is based on collaboration among elected repositories. The aim for efficiency of the discovery process in terms of response time and generated traffic implies querying repositories that are the most likely to store service advertisements that do match a requested service. Towards this goal, we use repository categorization as introduced in [22], which gives a compact overview of the repository content, however enhancing it with semantic content representation. More precisely, we use Bloom filters for summarizing the content of a repository. The main idea is to compute a vector v of m bits per registry, which corresponds to a Bloom filter. For any advertised capability Adv, its semantic description relies on a set of ontologies $O(Adv) = \{O_1, O_2, ..., O_n\}$ for describing its inputs, outputs, category and properties. For each capability Adv provided by a networked service and stored in a repository, its description in terms of its used ontologies is hashed with k independent hash functions. Each ontology is considered in terms of its URI. The bits of the vector v whose positions are given by the results of the k hash functions are set to 1, i.e., the bits at position $h_1(O(Adv)), h_2(O(Adv)), ..., h_k(O(Adv))$ are set to 1.

In order to determine whether a repository possibly stores a requested capability Req, we check in the repository's Bloom filter whether the bit positions $h_1(O(Req))$, $h_2(O(Req))$, ..., $h_k(O(Req))$ are all set to 1. If there is a bit that is not set to 1, the repository does not contain the required capability. On the other hand, if all the bits are set to 1, the repository is likely to contain the required capability, and EASY local service discovery is performed in that repository. The probability of a false positive depends on the parameters k, which is the number of hash functions, and m, which is the size of the Bloom filter. These values can be chosen so that the probability of false positives is minimized.

The cooperation between repositories is performed by exchanging the Bloom filters that give an overview of the repositories' content. The exchange of Bloom filters is done when new repositories are elected, and as well reactively, i.e., requested by another repository when the percentage of false positives reaches a given threshold.

According to the deployment policy, each mobile node is associated to at least one repository. When the mobile node seeks a service characterized by a set of required capabilities, it sends a

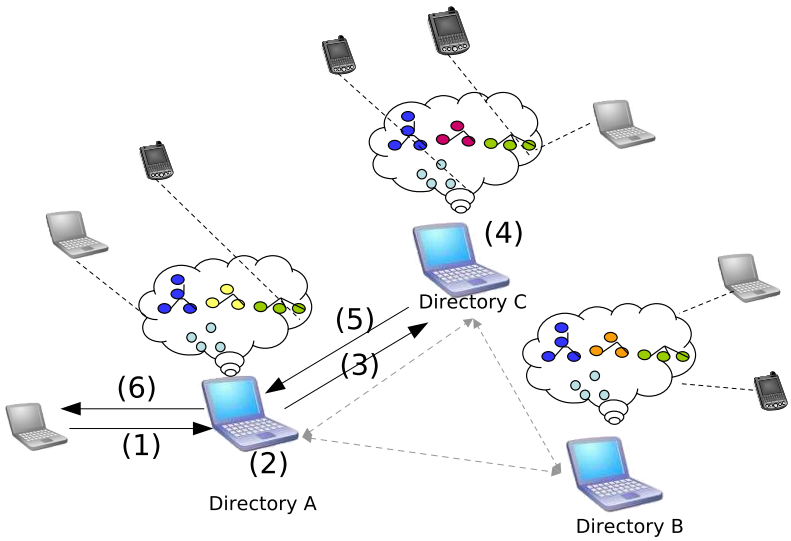


Figure 12: EASY-Ariadne

query message to the repository that is responsible for its vicinity. The repository performs for each required capability EASY local service discovery, as described in Section 4.3.2. If the required capabilities are not stored locally, the repository forwards the request to a subset of repositories that are likely to store capabilities that match the request. The repositories to which the request is forwarded are selected according to their Bloom filters.

Figure 12 provides an overview of the EASY-Ariadne architecture. In the figure, three nodes have been elected to act as repositories. When a service request is issued, the repository node that is in the vicinity of the service requester, i.e., repository A, receives the service request (Step (1)). The repository performs an EASY local service discovery to find capabilities that semantically match the capabilities of the requested service (Step (2)). Service advertisements providing these capabilities are returned to the requester. If some capabilities have not been found locally, another request is sent to remote repositories that are likely to store relevant capabilities according to their summarized description, i.e., Boom filters (Step (3)). These repositories perform an EASY local service discovery (Step (4)), and return the corresponding service advertisements (Step (5)), which are sent back to the requester.

The deployment of EASY on top of Ariadne has required the introduction of slight changes in the implementation of Ariadne, among which the update of the hashing procedure, i.e., hashing information from EASY-L instead of hashing WSDL descriptions, as well as extending the API of Ariadne in order to support the advertisement and matching of semantic descriptions. Nevertheless, introducing these slight changes in Ariadne enables performing efficient, rich, semantic, context- and QoS-aware service discovery and has further increased the scalability of Ariadne as discussed in Section 5.2.2.

Ontology	Classes	Max Ancestors	Caseau	Krall	Prime Max/Avg
SUMO	630	19	48	30	83 / 42
Wine & Food	133	14	39	33	53 / 23
Pizza	99	12	40	37	40 / 23
Gene Ontology	20945	54	2155	151	361 / 82
Java 1.30	5438	19	1568	68	112 / 31
OpenCyc	25565	89	1420	350	6817 / 272

Table 1: Comparison of encoding length of a single class in bits

5.2 Performance evaluation

For evaluating our solution we performed two main experiments. The first experiment aims at evaluating our solution for encoding multiple inheritance hierarchies using prime numbers. This evaluation is described is Section 5.2.1. The second experiment described in Section 5.2.2 evaluates the impact of introducing EASY on top of Ariadne.

5.2.1 Performance evaluation of the encoding algorithm

We have tested our proposed encoding mechanisms and subsumption heuristics on a set of multiple inheritance hierarchies, including: the Suggested Upper Merged Ontology [17]; the OpenCyc upper ontology[16]; several well-known ontology tutorial examples[17],[18]; the Gene Ontology[19], which provides a vocabulary of genes from any organism; and the Java 1.30 types hierarchy, which is part of a subtyping benchmark[20]. Table 1 provides an overview of the encoding lengths achieved by various existing algorithms (see Section 2.4.1) and our algorithm. The results for existing algorithms show the largest encoding length for a class in the hierarchy, expressed in bits. For the binary matrix method, this is equal to the size of the hierarchy. For our prime-based algorithm, the last column shows: the largest encoding lengths for the heuristic that minimizes the largest encoding length; and the average encoding length for the heuristic that minimizes the total encoding length. Besides achieving conflict-free incremental encoding, the encoding lengths produced by our algorithm are comparable if not better than the ones of existing algorithms. A complete description of our encoding and subsumption algorithms, along with other heuristic-based optimizations, and comparison of performance results with other techniques are discussed in detail in [20].

5.2.2 Performance of EASY-Ariadne

We have implemented a prototype of EASY-Ariadne for evaluating the impact of introducing semantic service matching in Ariadne, which originally uses basic WSDL-based syntactic matching of Web services. We have performed our evaluations on a Toshiba Satellite notebook with a 1.6 GHz Intel Centrino processor and 512 MB of RAM. In all the experiments that we performed, we increased the number of services from 1 to 100. The service descriptions given in EASY-L are using 22 different ontologies, and each service description contains a single provided capability. Note that for all our experiments each value is calculated from an average of five runs. Figure 13 shows the results of our first experiment, which evaluates the time to create graphs of services in an empty repository. A scenario for this experiment would be realized when a repository leaves the network and when another one is elected and has to host the set of service descriptions available in its vicinity. Figure 13 shows three measurements: (1) the time

[16] OpenCyc: http://www.opencyc.org/
[17] OWL Guide: http://www.w3.org/TR/owl-guide/
[18] Pizza Ontology: http://www.co-ode.org/ontologies/pizza/
[19] Gene Ontology: http://www.geneontology.org/
[20] Java subtyping benchmarks: http://www.zibin.net/subtyping-benchmarks.html

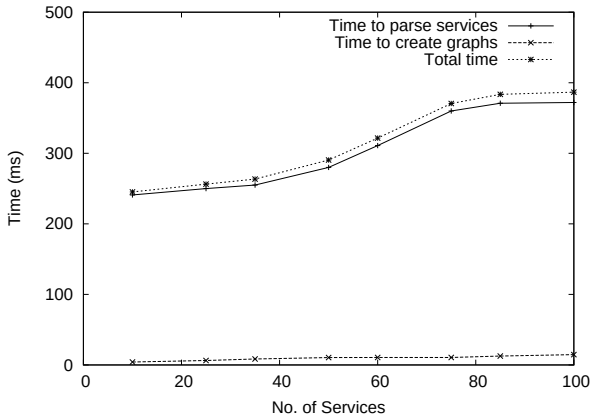


Figure 13: Time to create graphs

to parse the service descriptions; (2) the time to organize the service capabilities into graphs; and (3) the total time, i.e., time to parse and create the graphs. From this figure, we notice that the time to create the graphs is negligible compared to the time to parse service descriptions, i.e., XML parsing time, which is mandatory due to the use of Web services and Semantic Web technologies.

The results given by the second experiment that we performed are depicted in Figure 14. This experiment shows the time to insert a new service advertisement in a repository. This figure shows 3 measurements: (1) the time to parse the EASY-L description of the new service; (2) the time to classify the service capabilities within the repository graphs; and (3) the total time, i.e., the time to parse and classify the service capabilities. Results show that the time to classify service capabilities in a set of existing graphs is negligible compared to XML parsing time of the service description. We also notice that this time is nearly constant. This is due to the fact that the number of semantic matchings performed in the repository in order to insert a capability depends neither on the total number of services on the repository nor on the number of graphs. The time to insert a capability depends on the number of root and leaf nodes in the repository graphs as well as the number of capabilities contained in the graph in which the capability will be inserted. This is due to the fact that graphs are indexed using the ontologies that are being used in the capabilities' descriptions, which allows pre-selecting a subset of graphs that are likely to be appropriate for the insertion of the new capability. Thus, only a few number of semantic matches are performed in order to insert a capability in a repository.

The results of the third experiment that we performed are depicted in Figure 15. In this experiment, we evaluate the time to match a service request with services hosted by a repository. Furthermore, we compare the time to match a request in an organized repository with the time to match a request in an unorganized repository. Results are given without the XML parsing time of the request description. In this figure, we notice that without repository organization, the average overhead for matching is around 50% of the time to match when the repository is organized. Moreover, we notice that the time to match a request in the classified repository is nearly constant, which is due to the graphs indexing and the repository structuring. We also notice that the response time to match a required capability, excluding XML parsing time, is in the order of few milliseconds.

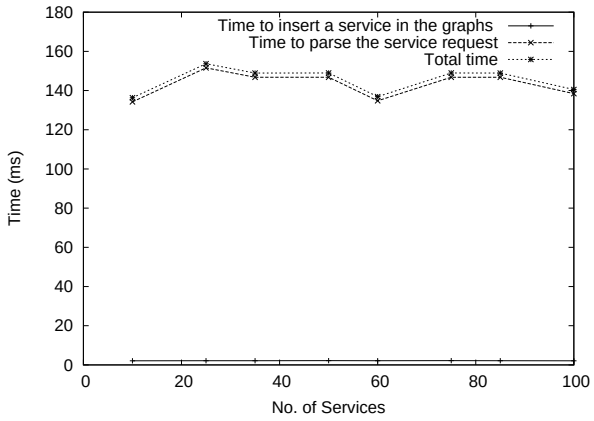


Figure 14: Time to publish a service advertisement

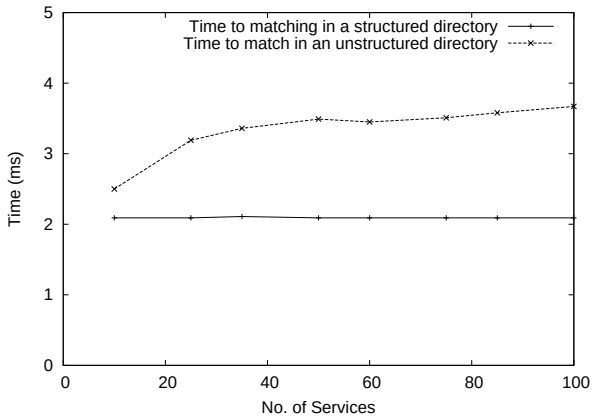


Figure 15: Time to match a service request: Organized vs Unorganized repository

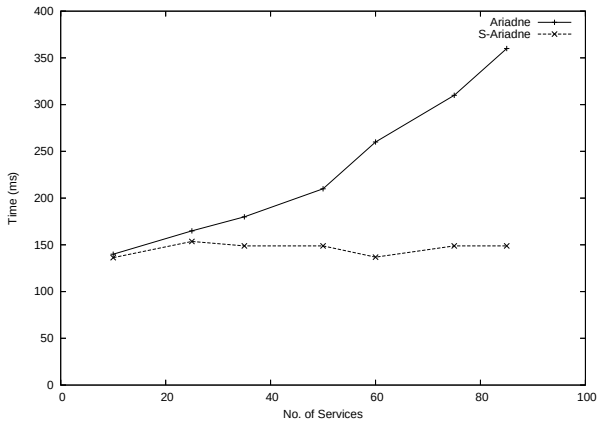


Figure 16: Time to match a service request: Ariadne vs EASY-Ariadne

The last experiment that we performed is a comparison of the response time given by the classical syntactic-based matching performed by Ariadne and the optimized semantic matching performed by EASY-Ariadne. The results are given in Figure 16. This figure shows that the response time given by Ariadne is increasing with the number of services available in the repository, while EASY-Ariadne has an almost stable response time, which is due to the following reasons: (1) using EASY-Ariadne, the services are parsed once at the publishing phase and their capabilities are classified, which avoids matching a request with all the services of the repository; (2) due to the numeric encoding of ontologies, the semantic matching performed by EASY-Ariadne reduces to a numeric comparison of codes, while using Ariadne the matching is performed by syntactically comparing the WSDL descriptions. We conclude that, using EASY-Ariadne, semantic matching, which allows to leverage the openness of pervasive computing environments, can be performed more efficiently than classical syntactic matching. Furthermore, thanks to repository indexing and structuring, EASY-Ariadne is more scalable than Ariadne.

6 Conclusion

The pervasive computing vision is increasingly enabled by the large success of wireless networks and devices. In pervasive environments, heterogeneous software and hardware resources may be discovered and integrated transparently towards assisting the performance of users' daily tasks. An essential requirement towards the realization of such a vision is the availability of mechanisms enabling the discovery of resources that best fit the client applications' needs among the heterogeneous resources that populate the pervasive environment. Based on the Service Oriented Architecture (SOA) paradigm, which allows the abstraction of the heterogeneous software and hardware resources as services described using structured service description languages, a number of service discovery protocols (SDPs) have emerged. However, these protocols rely on the syntactic conformance of service interfaces, which requires a common agreement on the syntax underlying the specification of such interfaces world-wide; this is hardy achievable in open pervasive environments. Furthermore, these SDPs provide limited support of service context and QoS properties, which is a key requirement towards the realization of the user-centric vision

aimed at by the pervasive paradigm. Building upon semantic Web technologies, and particularly ontologies, allows the unambiguous semantic, context and QoS specification of services in pervasive computing environments. However, such rich specifications require the use of costly semantic reasoning on the employed ontologies in order to assess the conformance of service capabilities against client requests.

Considering the large number of SDPs already deployed in pervasive environments, the objective of this article was not to propose yet another SDP, but to introduce a comprehensive solution to efficient, semantic, context- and QoS-aware service discovery, which can easily be deployed on top of existing SDPs. We first introduced EASY-Language (shortly EASY-L), a language for the semantic, context and QoS specification of service capabilities, and its corresponding set of conformance relations (EASY-Matching, shortly EASY-M). EASY-L is a simple and extensible language specified in OWL, which captures the essential information necessary for matching functional and non-functional properties of pervasive services. Furthermore, EASY-M defines three relations for matching service functional capabilities and allows rating services with respect to their suitability for a specific request. Moreover, EASY-M provides the means for selecting the service that best fits the non-functional requirements of service clients by taking into account client preferences among the various, heterogeneous properties. Based on EASY-L and EASY-M, EASY performs efficient service discovery on top existing SDPs thanks to two main optimizations. First, EASY relies on the offline encoding of classified ontology hierarchies, which allows reducing the costly semantic reasoning on ontologies to a numeric comparison of codes. Our encoding algorithm, which relies on prime numbers, supports incremental, conflict-free encoding, which allows freely reusing and extending existing ontologies. Moreover, compared to existing encoding algorithms, our solution proved satisfactory in terms of the employed code lengths. Second, contrary to existing approaches to efficient semantic service discovery that opt for overloading the service advertisement phase in order to gain efficiency in the service request phase, EASY performs both efficient service advertisement and service request. Indeed, EASY benefits from the aforementioned encoding technique for efficiently organizing semantic service specifications in service repositories or caches. This organization of service registries (caches) enables considerably reducing the number of semantic matchings performed to add a new service advertisement, as well as the number of matches performed to resolve a service request. To evaluate the flexibility and scalability of EASY, we further elaborated EASY-Ariadne, a prototype implementation of EASY on top of Ariadne, which is a scalable semi-distributed Web service discovery protocol for MANETs. Experimental results show that with slight changes introduced in Ariadne, EASY-Ariadne service providers and clients are provided with support for efficient rich, context- and QoS-aware service discovery enabled by EASY. Moreover, thanks to the encoding and organizing of semantic service specifications, EASY-Ariadne performs better than its ancestor Ariadne, and is further more scalable. Our future work include the deployment of EASY on top of MUSDAC[21], a middleware for multi-network, multi-protocol service discovery and access, in order to enable efficient semantic matching on top of the various SDPs supported by MUSDAC (e.g., UPnP, SLP).

Acknowledgments

This research is partially supported by the European IST AMIGO project[22] (EU-IST-004182).

[21] MUSDAC: http://www-rocq.inria.fr/arles/download/ubisec/

[22] Amigo Project: http://www.hitech-projects.com/euprojects/amigo/

References

[1] R. Agrawal, A. Borgida, H. V. Jagadish, Efficient management of transitive relationships in large data and knowledge bases, in: Proceedings of the 1989 ACM SIGMOD international conference on Management of data(SIGMOD '89), 1989.

[2] H. Ait-Kaci, R. S. Boyer, P. Lincoln, R. Nasr, Efficient implementation of lattice operations, Programming Languages and Systems 11 (1) (1989) 115–146.

[3] Amigo Consortium, Detailed design of the amigo middleware core, Project Deliverable D3.1b. (2005).

[4] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider (eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003 (2003).

[5] S. Ben Mokhtar, A. Kaul, N. Georgantas, V. Issarny, Efficient semantic service discovery in pervasive computing environments, in: Proceedings of ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'06), 2006.

[6] S. Ben Mokhtar, A. Kaul, N. Georgantas, V. Issarny, Towards efficient matching of semantic web service capabilities, in: Proceedings of the workshop of Web Services Modeling and Testing (WS-MATE'06), 2006.

[7] T. Berners-Lee, J. Hendler, O. Lassila, The semantic web, Scientific American, 2001.

[8] Y. Caseau, Efficient handling of multiple inheritance hierarchies, in: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA'93), 1993.

[9] Y. Caseau, M. Habib, L. Nourine, O. Raynaud, Encoding of multiple inheritance hierarchies and partial orders, Computational Intelligence 15 (1999) 50–62.

[10] I. Constantinescu, B. Faltings, Efficient matchmaking and directory services, in: Proceedings of the IEEE/WIC International Conference on Web Intelligence (WI'03), 2003.

[11] A. K. Dey, G. D. Abowd, Towards a better understanding of context and context-awareness, in: Workshop on The What, Who, Where, When, and How of Context-Awareness, Conference on Human Factors in Computer Systems (CHI'01), 2001.

[12] J. G. P. Filho, M. van Sinderen, Web service architectures - semantics and context-awareness issues in web services platforms, Tech. rep., Telematica Instituut (2003).

[13] J. M. Hellerstein, J. F. Naughton, A. Pfeffer, Generalized search trees for database systems, in: Proceedings of the 21st International Conference of Very Large Data Bases, VLDB'95, 1995.

[14] A. Krall, J. Vitek, N. Horspool, Near optimal hierarchical encoding of types, in: 11th European Conference on Object Oriented Programming (ECOOP'97), Springer, 1997.

[15] J. Liu, V. Issarry, QoS-aware service location in mobile ad-hoc networks, in: IEEE International Conference on Mobile Data Management (MDM'04), 2004.

[16] S. Majithia, D. W. Walker, W. A. Gray, A framework for automated service composition in service-oriented architecture, in: 1st European Semantic Web Symposium, 2004.

[17] I. Niles, A. Pease, Towards a standard upper ontology, in: Proceedings of the international conference on Formal Ontology in Information Systems(FOIS'01), 2001.

[18] M. Paolucci, T. Kawamura, T. R.Payne, K. Sycara, Semantic matching of Web services capabilities, Lecture Notes in Computer Science 2342 (2002) 333–347.

[19] M. P. Papazoglou, D. Georgakopoulos, Special section in Communications of the ACM, chap. Service-oriented computing, ACM Press, 2003.

[20] D. Preuveneers, Y. Berbers, Prime numbers considered useful: Ontology encoding for efficient subsumption testing, Tech. Rep. CW464. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW464.abs.html, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (October 2006).

[21] D. Preuveneers, J. V. den Bergh, D. Wagelaar, A. Georges, P. Rigole, T. Clerckx, Y. Berbers, K. Coninx, V. Jonckers, K. D. Bosschere, Towards an extensible context ontology for ambient intelligence., in: EUSAI, 2004.

[22] F. Sailhan, V. Issarny, Scalable service discovery for MANET, in: Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom'05), 2005.

[23] M. Satyanarayanan, Pervasive computing: Vision and challenges, IEEE Personal Communications 08 (4) (2001) 10–17.

[24] L. K. Schubert, M. A. Papalaskaris, J. Taugher, Determining type, part, color, and time relationships, IEEE Computer 16 (10) (1983) 53–60.

[25] E. Sirin, B. Parsia, J. Hendler, Template-based composition of semantic web services, in: AAAI Fall Symposium on Agents and the Semantic Web, 2005.

[26] N. Srinivasan, M. Paolucci, K. Sycara, Adding owl-s to uddi, implementation and throughput, in: Proceedings of the Workshop on Semantic Web Service and Web Process Composition, 2004.

[27] K. Sycara, J. Lu, M. Klusch, S. Widoff, Matchmaking among heterogeneous agents on the internet, in: Proceedings of the 1999 AAAI Spring Symposium on Intelligent Agents in Cyberspace, 1999.

[28] K. Sycara, M. Paolucci, A. Ankolekar, N. Srinivasan, Automated discovery, interaction and composition of semantic web services, Web Semantics: Science, Services and Agents on the World Wide Web, 2003.

[29] The DAML Services Coalition, Bringing semantics to web services: The owl-s approach, in: Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC'04), 2004.

[30] D. Trastour, C. Bartolini, J. Gonzalez-Castillo, A semantic web approach to service description for matchmaking of services, in: Proceedings of the first Semantic Web Working Symposium, (SWWS), 2001.

[31] L.-H. Vu, M. Hauswirth, K. Aberer, Towards p2p-based semantic web service discovery with qos support., in: Business Process Management Workshops, 2005.

[32] A. M. Zaremski, J. M. Wing, Specification matching of software components, ACM Transactions on Software Engineering and Methodology, 1997.

[33] A. M. Zaremski, J. M. Wing, Signature matching: a tool for using software libraries, ACM Transactions on Software Engineering and Methodology 4 (2) (1995) 146–170.

[34] F. Zhu, M. W. Mutka, L. M. Ni, Service discovery in pervasive computing environments, Pervasive Computing, IEEE 4 (4) (2005) 81–90.

[35] Y. Zibin, J. Gil, Efficient subtyping tests with PQ-encoding, in: Conference on Object-Oriented, 2001.

A Proofs

We recall the definitions of the three matching relations used hereafter in this appendix:

ExactCapabilityMatch(*Adv, Req*) =
$\forall$ *in* $\in$ *Req*.In, $\exists$ *in'* $\in$ *Adv*.In: **ConceptMatch**(*in', in*) = exact and
$\forall$ *out* $\in$ *Req*.Out, $\exists$ *out'* $\in$ *Adv*.Out: **ConceptMatch**(*out', out*) = exact and
ConceptMatch(*Adv.Category, Req.Category*) = exact

InclusiveCapabilityMatch(*Adv, Req*) =
$\forall$ *in* $\in$ *Req*.In, $\exists$ *in'* $\in$ *Adv*.In: **ConceptMatch**(*in', in*) = exact|plugin and
$\forall$ *out* $\in$ *Req*.Out, $\exists$ *out'* $\in$ *Adv*.Out: **ConceptMatch**(*out', out*) = exact|plugin and
ConceptMatch(*Adv.Category, Req.Category*) = exact|plugin

WeakCapabilityMatch(*Adv, Req*) =
$\forall$ *in* $\in$ *Req*.In, $\exists$ *in'* $\in$ *Adv*.In: **ConceptMatch**(*in', in*) $\neq$ fail and
$\forall$ *out* $\in$ *Req*.Out, $\exists$ *out'* $\in$ *Adv*.Out: **ConceptMatch**(*out', out*) $\neq$ fail and
ConceptMatch(*Adv.Category, Req.Category*) $\neq$ fail

Note that by definition:
Prop 0: **ExactCapabilityMatch**(*Adv, Req*) $\Rightarrow$ **InclusiveCapabilityMatch**(*Adv, Req*) $\Rightarrow$ **WeakCapabilityMatch**(*Adv, Req*).

A.1 Proof of property (Prop 1)

Prop 1: $\neg$ **InclusiveCapabilityMatch**($Root_i$, *Adv*): $Root_i \in$ **Roots**(*G*) $\Rightarrow$
$\forall$ *C* $\in$ **Successors**($Root_i$): $\neg$ **InclusiveCapabilityMatch**(*C, Adv*)

We prove (Prop 1) by contradiction. Assume $\neg$ (Prop 1), i.e.:

$\neg$ **InclusiveCapabilityMatch**($Root_i$, *Adv*): $Root_i \in$ **Roots**(*G*) and (1)
$\neg$ ($\forall$ *C* $\in$ **Successors**($Root_i$): $\neg$ **InclusiveCapabilityMatch**(*C, Adv*)) (2)

(2) $\Leftrightarrow$ $\exists$ *C* $\in$ **Successors**($Root_i$): **InclusiveCapabilityMatch**(*C, Adv*)
On the other hand: *C* $\in$ **Successors**($Root_i$) $\Rightarrow$ **InclusiveCapabilityMatch**($Root_i$, *C*) from the definition of the function **Successors**(); thus:

(2) $\Leftrightarrow$ $\exists$ *C* $\in$ **Successors**($Root_i$): **InclusiveCapabilityMatch**($Root_i$, *C*) and **InclusiveCapabilityMatch**(*C, Adv*)

From the transitivity property of the function **InclusiveCapabilityMatch**(), we have: (2) $\Leftrightarrow$ **InclusiveCapabilityMatch**($Root_i$, *Adv*)

Replacing (2) in the list of our assumptions with this equivalence results into:
$\neg$ **InclusiveCapabilityMatch**($Root_i$, *Adv*) and **InclusiveCapabilityMatch**($Root_i$, *Adv*). This can never be true, and therefore, the assumption is false and (Prop 1) is true.

A.2 Proof of property (Prop 2)

Prop 2: ¬ **InclusiveCapabilityMatch**(*Adv, Leaf*$_i$): *Leaf*$_i$ ∈ **Leaves**(*G*) ⇒
∀ *C* ∈ **Predecessors**(*Leaf*$_i$): ¬ **InclusiveCapabilityMatch**(*Adv, C*)

We prove (Prop 2) by contradiction. Assume ¬ (Prop 2), i.e.:

¬ **InclusiveCapabilityMatch**(*Adv, Leaf*$_i$): *Leaf*$_i$ ∈ **Leaves**(*G*) and (1)
¬ (∀ *C* ∈ **Predecessors**(*Leaf*$_i$): ¬ **InclusiveCapabilityMatch**(*Adv, C*)) (2)

(2) ⇔ ∃ *C* ∈ **Predecessors**(*Leaf*$_i$): **InclusiveCapabilityMatch**(*Adv, C*)
On the other hand: *C* ∈ **Predecessors**(*Leaf*$_i$) ⇔ **InclusiveCapabilityMatch**(*C, Leaf*$_i$) from the definition of the function **Predecessors**(); thus:

(2) ⇔ ∃ *C* ∈ **Predecessors**(*Leaf*$_i$): **InclusiveCapabilityMatch**(*Adv, C*) and **InclusiveCapability-Match**(*C, Leaf*$_i$)

From the transitivity property of the function **InclusiveCapabilityMatch**(), we have: (2) ⇔ **InclusiveCapabilityMatch**(*Adv, Leaf*$_i$)

Replacing (2) in the list of our assumptions with this equivalence results into:
¬ **InclusiveCapabilityMatch**(*Adv, Leaf*$_i$) and **InclusiveCapabilityMatch**(*Adv, Leaf*$_i$). This can never be true, and therefore, the assumption is false and (Prop 2) is true.

A.3 Proof of property (Prop 3)

Prop 3: ∀ *Root*$_i$ ∈ **Roots**(*G*): ¬ **WeakCapabilityMatch**(*Root*$_i$, *Req*) ⇒
∀ *C* ∈ G: ¬ **WeakCapabilityMatch**(*C, Req*)
We prove (Prop 3) by contradiction. Assume ¬ (Prop 3), i.e.:

∀ *Root*$_i$ ∈ **Roots**(*G*): ¬ **WeakCapabilityMatch**(*Root*$_i$, *Req*) and(1)
¬ (∀ *C* ∈ *G*: ¬ **WeakCapabilityMatch**(*C, Req*)) (2)

(2) ⇔ ∃ *C* ∈ *G*: **WeakCapabilityMatch**(*C, Req*). On the other hand:
(*C* ∈ *G*) ⇔ (∃ *R* ∈ **Roots**(*G*): *C* ∈ **Successors**(*R*)). From the definition of **Successors**():
⇔ (∃ *R* ∈ **Roots**(*G*): **InclusiveCapabilityMatch**(*R, C*)). After applying (Prop 0):
⇔ (∃ *R* ∈ **Roots**(*G*): **WeakCapabilityMatch**(*R, C*))

Thus (2) becomes:
(2) ⇔ ∃ *C* ∈ *G*, ∃ *R* ∈ **Roots**(*G*): **WeakCapabilityMatch**(*C, Req*) and **WeakCapability-Match**(*R, C*)
From the transitivity property of the relation **WeakCapabilityMatch**() we have:
(2) ⇔ ∃ *R* ∈ **Roots**(*G*): **WeakCapabilityMatch**(*R, Req*), which contradicts (1). Thus the assumption is false and (Prop 3) is true.