Distrib Parallel Databases (2007) 21:5–37
DOI 10.1007/s10619-006-7003-7

Context-based matching for Web service composition

Brahim Medjahed · Yacine Atif

Published online: 14 November 2006
© Springer Science + Business Media, LLC 2007

Abstract In this paper, we propose a novel matching framework for Web service composition. The framework combines the concepts of Web service, context, and ontology. We adopt a broad definition of context for Web services, encompassing all information needed for enabling interactions between clients and providers. Context-based matching for Web services requires dealing with three major research thrusts: context categorization, modeling, and matching. We first propose an ontology-based categorization of contextual information in Web service environments. We then define a two-level mechanism for modeling Web service contexts. In the first level, service providers create context specifications using category-specific Web service languages and standards. In the second level, context specifications are enveloped by policies (called context policies) using WS-Policy standard. Finally, we present a peer-to-peer architecture for matching context policies. The architecture relies on a context matching engine, context policy assistants, and context community services. Community services implement rule-based techniques for comparing context policies.

Keywords Web service composition · Matching · Context · Policy · Ontology · Agent

Recommended by: Djamal Benslimane and Zakaria Maamar

B. Medjahed (✉)
Department of Computer and Information Science, University of Michigan, Dearborn, USA
e-mail: brahim@umich.edu

Y. Atif
Institute of Information Sciences and Technology, Massey University, New Zealand
e-mail: Y.Atif@massey.ac.nz

1 Introduction

The emergence of Web services has stimulated organizations from different domains (e.g., business, government) to provide access to their core applications on the Web [1]. Web services expose XML-based interfaces (such as WSDL interfaces [11]) that can be programmatically invoked on the Web. Two main motivations are promoting Web services as the technology of choice for integrating intra and inter-enterprise applications. The use of standard technologies (e.g., XML, HTTP) reduces heterogeneity, and is therefore key to facilitating application integration [1]. Additionally, the adoption of a document-based messaging model in Web services caters for loosely coupled relationships among cross-organizational applications [36]. Web services can be combined across companies to define *value-added* services [48]. For example, a `tax preparator` service may be defined by combining financial services at employees companies to get W2 form (commonly used in the US to list an employee's wages and tax withheld), banks and investment companies services to retrieve investment information, and electronic tax filing services provided by state and federal revenue agencies [37]. We refer to a service that combines functionalities provided by other Web services as a *composite service* (e.g., `tax preparator`). The Web services that build-up a composite service (e.g., bank Web service) are called *participants*.

The process of composing Web services (i.e., *service composition*) is far from trivial [1, 39]. To support users (or composers) in this effort, *composition engines* are being developed. Composition engines provide abstractions and mechanisms facilitating the definition and execution of composite services. Standardization efforts are under way in the service composition area. One such effort is BPEL (Business Process Execution Language for Web Services), a language for the specification of composite services [25]. BPWS4J, ActiveBPEL, Collaxa, and Oracle BPEL Process Manager are examples of existing BPEL composition engines. Developing end-to-end composition engines requires dealing with several research thrusts such as orchestration, exception handling, transactions, and service matching [27]. Our focus in this paper is on *service matching* for Web service composition.

Service matching refers to the process of comparing Web services based on their capabilities. The aim is to determine whether a Web service "relates to" another Web service where "relates to" generically stands for "is equivalent to" and "is compatible with". We identify two scenarios where service matching can be used in a service composition engine: *substitution* and *dynamic composition*. *Substitution* aims at replacing a participant by another Web service. This may happen because of a network problem or if the original participant is made temporarily unavailable by its provider (e.g., for maintenance or upgrade). Such substitution should be done in a way that will not affect the business logic and behavior of the entire composite service. *Dynamic composition* enables the specification of composite services without knowing *a priori* which participants will be actually used at run-time. Composers give abstract specifications of their participants (e.g., by using the notion of generic service node defined in eFlow [6]). At run-time, the composition engine selects Web services that match participants' abstract specifications.

Several techniques have been proposed to deal with service matching [3, 7, 8, 21, 28, 35, 41, 42, 44, 47]. However, these techniques use a limited set of matching attributes defined within service descriptions (e.g., DAML-S descriptions [34]).

They merely compare text descriptions, signatures (inputs and outputs), and logical constraints about inputs and output. In this paper, we provide a generic matching framework for service composition. The framework uses Web service *contexts* to process matching requests submitted by the composition engine. Simply put, a *context* is "*any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves*" [13]. Context has been used in several areas such as machine learning, computer vision, information retrieval, decision support, and pervasive computing [40]. Our aim in this paper is to apply context to service matching. We view context as any client or provider-related information that enables interactions between service clients and providers. Our main contributions in this paper are summarized as follows:

- We propose a novel *categorization* of Web service contexts suitable to service matching. We use the concept of *ontology* to define such categorization. An ontology is a formal and explicit specification of a shared conceptualization [15]. The proposed categorization is dynamic in a sense that new context categories may be added on the fly.
- We define a *two-level* mechanism for *modeling* Web service contexts. In the first level, service providers create *context specifications* using appropriate Web service languages and standards (e.g., WS-security, WS-Agreement). In the second level, providers envelop their context specifications by *policies* using WS-Policy standard [22].
- We propose a *peer-to-peer* approach for matching Web service contexts. The approach relies on a *context matching engine*, *context policy assistants*, and *context community services*. Community services implement *rule*-based techniques for comparing context policies.

The rest of the paper is organized as follows. Section 2 introduces a scenario from the bioinformatics domain to motivate our approach. Section 3 describes the proposed context-aware Web service model. Section 4 gives details about the framework for matching Web services contexts. Section 5 is devoted to the implementation of the proposed approach. Section 6 gives an overview of the related work. Section 7 provides concluding remarks.

2 Motivating scenario

The motivating scenario is related the analysis of protein sequence information in the bioinformatics domain. Consider a Gigabit Ethernet environment linking several bioinformatics institutions. Each of the contributing institutions has an entry point to this service grid to conduct scientific activities by invoking bioinformatics Web services. Access to the service grid is provided to authenticated biologists through a Web BioPortal. Such an infrastructure helps scientists avoid manual maintenance and execution of several Web service-enabled bioinformatics applications. Performing a complex process such as protein identification requires the combination of several bioinformatics Web services. BioPortal uses a service composition engine to handle

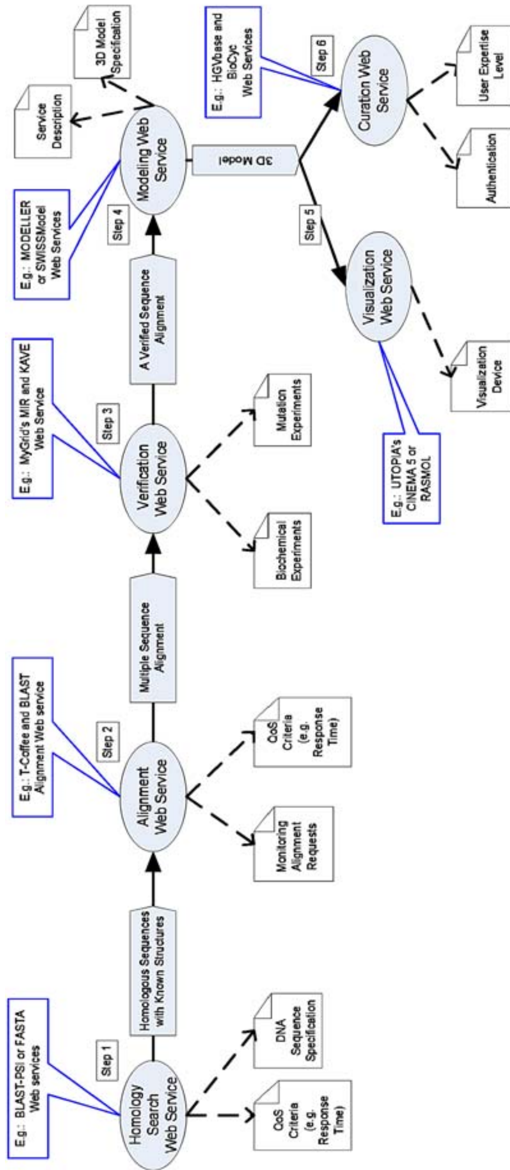

Fig. 1 Motivating scenario

the orchestration and management of such services. Figure 1 depicts an example of a composite service used for analyzing DNA sequences.

The biologist first submits the DNA sequence specification to the BioPortal. The BioPortal interacts with the service registry (e.g., UDDI) to discover a relevant `homology search` Web service (Step 1). Homology search refers to scouring a sequence database to find sequences that are likely to be homologous (i.e., have a common ancestor) to a given sequence [20]. Contextual information of `homology search` includes quality of service (e.g., response time) and constraints about the sequence specification. BLAST-PSI and FASTA are examples of homology search services. The execution of `homology search` generates a set of the target-homologous protein sequences which genes' data are available. An `alignment` Web service follows to narrow down the search (Step 2). Alignment refers to the use of amino-acid data to determine the degree of base or amino acid similarity which reveals the degree of similarity between the target and the homologous genes [20]. T-Coffee and BLAST are examples of alignment services. The context of `alignment` Web service includes the specification of monitoring requests to check the status of alignment requests (e.g., estimated left time).

A large number of sequences may be aligned as similar to the target in the second step. To narrow down the search space, additional criteria in the target's specification are used. Such criteria refer to prior experimental results conducted on the target (Step 3). This task is performed by a `verification` Web service such as ^{my}Grid's MIR and KAVE services. The `verification` Web service compares protein experimental reactions (e.g., biochemical and mutation experiments) and returns appropriate results to determine whether there are proteins that lead to similar experimental results as the target. Contextual information includes the specification of experimental results. The resulting proteins which succeed the `verification` service are then analyzed and used to infer a model for the target (Step 4). The `modeling` Web service provides a three-dimensional (3D) model of the target based on homologous sequences. MODELLER and SWISSModel are examples of such services. Contextual information includes a service description (i.e., how to invoke it) and the specification language of the 3D model. The resulting model is displayed using a `visualization` Web service (Step 5) such as CINEMA 5 and RASMOL. This service is constrained by the user's graphical device capabilities which constitute part of the `visualization` service context. Concurrently with the visualization process, the identified target is published in a curation database through a `curation` Web service such as HGVbase and BioCyc (Step 6). Curation is the process of tracking the provenance of bioinformatics results to accurately describe the purpose and design of bioinformatics data [20]. The service uses as a context the biologist's expertise level and authentication information.

Assume now that the `curation` Web service used by BioPortal's composition engine is unavailable (e.g., because of network problems). To provide uninterrupted services to biologists, BioPortal needs to replace the existing `curation` service with another Web service that "matches" the former. Performing the matching process manually is tedious and time-consuming. In the rest of this paper, we propose to automate this process through the development of a context-based matching framework for Web service composition.

3 A context-aware Web service model

In this section, we describe the main components of the proposed model for matching context-aware Web services. In Section 3.1, we give a context-oriented definition of Web services. In Section 3.2, we propose a categorization of Web service contexts using RDF ontology language. In Section 3.3, we define a two-level mechanism for modeling Web service contexts. In Section 3.4, we illustrate the way contexts are organized into communities based on their categories. In Section 3.5, we introduce context-based policy assistants as means for facilitating the creation of contexts.

3.1 Web service = {context definitions}

A variety of definitions of the Web service concept are given by different industry leaders, research groups, and Web service consortia. Existing definitions range from the very generic and all-inclusive to the very specific and restrictive [1]. They mostly focus on the rationale behind Web services (e.g., interoperation) and major technologies for interacting with Web services (e.g., XML, SOAP, WSDL). For example, the W3C consortium defines a Web service as "*a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards*". In this section, we give a *context-oriented* definition of a Web service, focusing on contextual information available to service clients and providers during their interactions.

Web services involve two major participants (Fig. 2): *providers* and *clients*. Providers are the parties (e.g., businesses) that define and make services available on the Web. Clients are the entities that invoke those Web services. A client may be an end-user, software agent, another Web service, or a composition engine. Interactions between service providers and clients follow the producer-consumer model; providers offer Web service that are consumed by clients. Although a client and provider are involved in the same interaction, each participant has its own view on that interaction. From a client's perspective, interacting with providers depends on the situation, or *context*, of the current Web service. The *provider-related* context contains meta-data about the provider (and its service). Examples of such meta-data include service description, service security policy, and quality of service. From a provider's

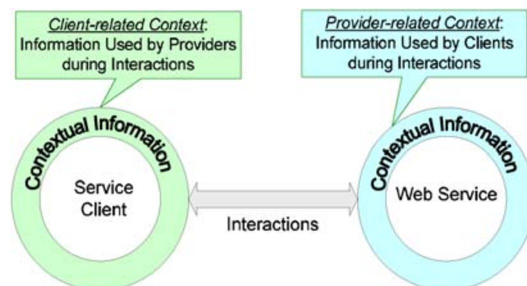

Fig. 2 Context-aware Web service interactions

perspective, interacting with a client depends on the situation, or *context*, of that client. The *client-related* context contains meta-data about the client. Examples of such meta-data include the user's location, expertise level (e.g., expert, novice), and identity.

We adopt a broad definition of context for Web services, encompassing all information needed for enabling interactions between clients and providers. As depicted in Fig. 2, we view a context as any information that can be used by (i) a Web service to interact with clients, and (ii) client to interact with Web services. Web services that use context, whether available to clients or providers, are called *context aware*. A *client-related* context includes information (about the client) used by the provider (or Web service) to interact with the client. The biologist's expertise level is an example of client-related context for the `curation` service (Fig. 1). A *provider-related* context includes information (about the Web service or provider) used by the client to interact with the Web service (or provider). In our motivating scenario, *Authentication* is an example of provider-related context for the `curation` service (Fig. 1). This document, attached to the `curation` service, specifies the way service clients are authenticated. It may for instance state that the service supports the "Basic 256 RSA 15" or the "3DES RSA 15" security algorithm suites [12]. Another example of provider-related context is a service description (e.g., in WSDL [11] or OWL-S [33]). This context describes the operational features of Web service (e.g., operation signature, messages, operation semantics). Details about the different categories of provider-related contexts are given in Section 3.2. We present below the definition of a context-aware Web service.

Definition . A *context-aware* Web service *WS* is defined by set of context definitions $\mathcal{C}(WS) = \mathcal{C}^C(WS) \cup \mathcal{C}^P(WS)$ where $\mathcal{C}^C(WS)$ and $\mathcal{C}^P(WS)$ are the sets of *client-related* and *provider-related* context definitions, respectively.

Our focus in the definition above is on context relevant to matching capabilities for Web service composition. Other types of contexts geared toward the execution and deployment of composite Web services may also be defined. We refer to these contexts as *run-time* contexts. I-context and W-context defined in [30] are examples of run-time contexts. I-context gives information about a specific running instance of a participant Web service (e.g., whether the instance is in-progress, suspended, or terminated). W-context contains information about all running instances of a participant Web service (e.g., number of service instances that are currently running). Information included in run-time contexts may be used in the matching process. For example, a matching engine may decide to replace a participant service WS_1 by an "equivalent" one, if the execution status of WS_1 (defined in the I-context) is "aborted". Handling run-time contexts is out of the scope of this paper.

3.2 Categorization of Web service contexts

The categorization of Web service contexts is important for the development of context-aware Web service environments. Despite the various attempts to develop a taxonomy for contexts, there is no generic context categorization. Relevant information differs from a domain to another and depends on the effective use of this

information [40]. We propose a categorization of contextual information in Web service environments. Figure 3 depicts our categorization as an RDF ontology graph [15]. One or more context definitions are associated to a Web service. Each context definition belongs to a certain category which can be either *client-related* or *provider-related*.

Client-related context represents the provider's view on the client. It is obtained either *explicitly* or *implicitly* [13]. For example, end-users may explicitly feed a Web service with inputs (e.g., name, address) that identify them. A `tour guide` Web service may display different customized information on a hand-held computer based on the implicitly-obtained tourist location. Both implicit and explicit contexts may have subcategories. For example, implicit context may be sensed (e.g., acquired via physical or software sensors) or derived (e.g., time and date). Details about client-related contexts are out of the scope of this paper. The *Provider-related* context models the client's view on the provider (or Web service). It is explicitly defined by the service provider. The provider of a Web service exposes a set of provider-related contexts to convey the conditions under which that service may be used or outsourced. Standards have so far been the key enablers for defining such contexts. For example, WSDL enables the specification of a service interface that includes the service operations, location, and protocols to access the service (e.g., SOAP/HTTP in RPC style). Clients use this context to decide whether or not to interact with a Web service.

We identify the following categories of provider-related contexts (Fig. 3): *functional*, *non-functional*, *domain*, and *valued-added* contexts. *Functional* context describes the operational features of a Web service (e.g., in OWL-S language). Functional context attributes are of two types: syntactic and semantic. Examples of syntactic attributes include the list of input/output parameters that define an operation's messages, the data types of these parameters, and the protocol to be used for invoking the Web service (e.g., SOAP/HTTP, ebXML Messaging Service). Examples of semantic attributes include the pre-condition and effect of an operation execution [37]. For example, the `modeling` Web service (Fig. 1) has a service description (i.e., functional context) that includes the SOAP/HTTP address to invoke the service. A *non-functional*, also called Quality of Service (QoS), context includes a set of metrics that measure the *quality* of a Web service. The international quality standard ISO 9000 describes *quality* as "the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs" [45]. Examples of such characteristics include time, availability, reliability, and cost. Two providers that support the same service functionalities may have different values for their qualitative attributes. These attributes model in fact the competitive advantage that providers may have on each other. For example, the `homology search` Web service has a non-functional context that includes an estimated response time for homology search requests.

The *domain* context is organized into domain-dependent sub-categories. Each application domain has its own requirements for interacting with Web services in that domain [36, 38, 53]. Shipping and billing are examples of contexts in B2B E-commerce. A context in bio-informatics may refer to an experiment description in an experiment specification language [53]. For example, the `verification` Web Service (Fig. 1) makes use of a range of biological contexts to elicit a target DNA sequence in hope of observing an expected phenotype. The *value-added* context brings "better" envi-

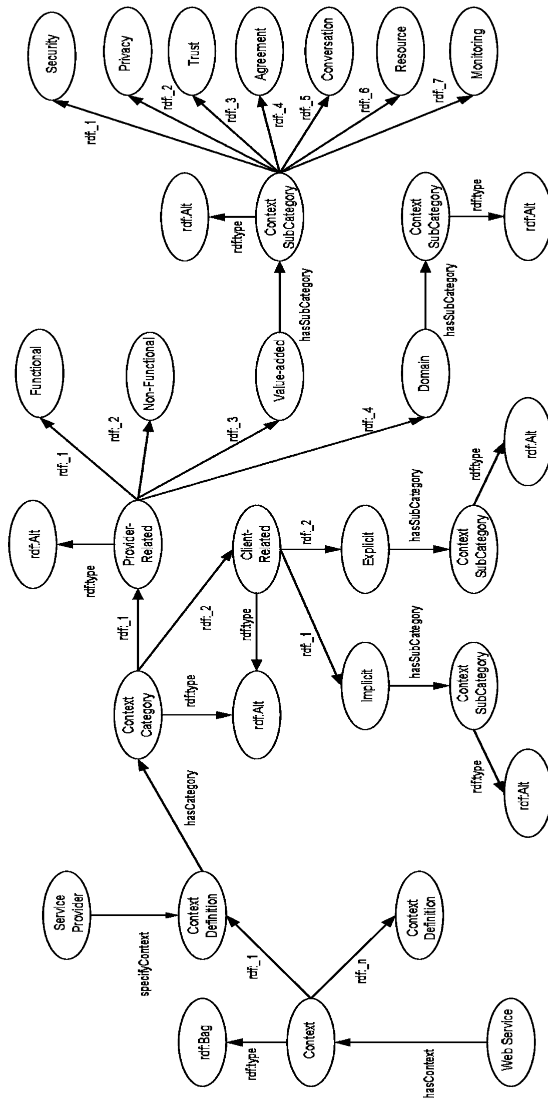

Fig. 3 RDF ontology for Web service context categories

ronments for Web service interactions. It includes a set of specifications (or context definitions) for supporting optional (but important) requirements (e.g., security, privacy) useful while interacting with a Web service. We organize value-added contexts into the following sub-categories:

- *Security*: A *security context* states whether a Web service is compliant with security requirements such as encryption, authentication, non-repudiation, and reliability [4]. The `curation` Web service (Fig. 1), for example, has an authentication context. Biologists need to be identified before storing their results in a curation database.
- *Privacy*: Providers indicate their privacy practices for a given Web service via *privacy contexts*. Examples of privacy practices include the list of external entities beyond the service provider where the collected data (e.g., input parameters) may be distributed and the purpose from collecting this data (e.g., marketing) [24]. Referring to Step 6 of our motivating scenario, the `curation` Web service may be restricted by the biologist's privacy requirements in revealing the undertaken DNA elicitation process.
- *Trust*: This context contains elements that are used to establish trust relationships between clients and providers [2]. In order to secure a client-provider communication, the two parties must exchange security credentials. However, each party needs to determine whether it can trust the asserted credentials of the other party. An illustration of this context is given by the `verification` Web service (Fig. 1) where comparative experimental reactions are performed to determine whether there are proteins whose matching contexts lead to similar biochemical evolution as the target. This step bears a certain level of trust requirement on the authority which provides the protein reactions' data.
- *Agreement* (*or Contracting*): Clients may require assurances or guarantees concerning the level and type of service being offered by the provider [48]. *Agreement contexts* enable the explicit specification of agreement terms between service providers and clients. For example, a disk storage service provider may specify the following agreement: "I guarantee 12 MB of memory to the client for one hour on weekdays. I have a high confidence (probability of 0.99) in meeting this guarantee. In the event, I am not able to meet this guarantee, I will pay a penalty of 50 US Dollars". Another example is the `homology search` Web service (Fig. 1) which provides QoS assurance guarantee on the response time to accomplish the requested service.
- *Conversation*: Conversation refers to the sequences of operations (i.e., message exchanges) that could occur between a client and a Web service as part of the invocation of that service [1]. Using a particular Web service typically involves performing sequences of operations in a particular order. The *conversation context* specifies the set of correct conversations that a client should support to interact with a Web service. For example, The `curation` Web service (Fig. 1) exposes a conversational context; biologists first provide authentication information. Then, they give information about their expertise level. Finally, they store their experiment data in the curation database.
- *Resource*: This context represents the computing means on which Web services operate. It gives the list of resources to be used while interacting with a Web service [16]. The *resource context* includes information about resources required from clients to invoke the Web service (e.g., wireless device). It also contains

properties (e.g., number of blocks, block size, and manufacturers) of a resource (e.g., disk) used by a Web service (e.g., a disk storage service). The `visualization` Web service in our motivating scenario may, for example, require that the client graphical interface would have the capacity to perform at a certain resolution level.
- *Monitoring*: This context is required for a number of purposes, including status checking, troubleshooting, performance tuning, and debugging [48]. The *monitoring context* offers mechanisms that check the status of a service invocation. It also includes elements for inquiring about the "health" of a service in real time by detecting signs of failure. For example, the `alignment` Web service (Fig. 1) may have a monitoring context that allows biologists to check the status of their alignment requests (e.g., estimated left time, current number of aligned sequences).

The domain category refers to *vertical* contexts that are valid in specific application domains such as B2B e-commerce, e-government, and bio-informatics. The other categories (i.e., functional, non-functional, and value-added) refer to *horizontal* contexts that are applicable across domains. The proposed ontology for Web service context categories is dynamic in the sense that new sub-categories may be added at any time. For example, negotiation may be added to the set of value-added contexts. Coordination and transaction sub-categories may be added under the conversation category. The security context may be organized into sub-categories encompassing contexts about authentication, encryption, non-repudiation, and reliability. New domain sub-categories may also be added at any time. The proposed technique for dynamically defining contexts is described in Section 4.3.

3.3 Modeling contexts as policies

Context modeling is an important issue that need to be addressed for enabling context-aware Web services. By context modeling, we refer to the language to be used for defining Web service contexts. The diversity of contextual information has led to several context modeling mechanisms such as *ConteXtML* [43], *contextual schemas* [49], *CxBR* (context-based reasoning) [18], and *CxG* (contextual graphs) [5]. These languages provide means for defining context is specific application domains such as pervasive computing, mobile computing, and robotics. They provide little or no support for defining context in Web service environments. In this section, we introduce an approach for context modeling suitable to service matching (Fig. 4).

Figure 4 depicts an onion-like structure of the proposed mechanism for modeling Web service contexts. At the center is the *category* level surrounded by the *envelope* level. In the category level, providers create specifications (called *context specifications*) using appropriate Web service languages and standards. Each context belongs to a certain category (as stated in Fig. 3). It is specified in a language related to that category. A *Type* attribute is associated to each context specification to state the language/standard used for that context. The use of a category level allows the leverage of existing Web service standards/languages for modeling contexts. It also ensures interoperability with "legacy" Web services; existing Web services may be exposed as context-aware and hence, be used in our framework. Our approach places no restrictions on the languages used in the category-specific level. Table 1 gives examples of standards/languages that may be adopted within each context category. For example,

Table 1 Examples of category-specific languages for Web service contexts

Category	Category-specific language
Functional	WSDL-S, DAML-S, and OWL-S
Non-Functional	WSCL (Web Services Conversation Language) and HQML(Hierarchical QoS Markup Language)
Security	WS-SecurityPolicy and WS-SecureConversation
Privacy	P3P-like language
Trust	WS-Trust
Agreement	ebXML's TPA and CPA, WS-Agreement, and Globus Toolkit's SLAs (Service Level Agreement)
Conversation	RosettaNet PIP (Partner Interface Processes) and WSCI (Web Service Choreography Interface)
Resource	WS-Resource
Monitoring	The Globus Toolkit's Monitoring and Discovery service
Domain	B2B E-commerce (shipping and billing) and bio-informatics (experiment specification language)

functional contexts may be specified in WSDL-S, DAML-S, or OWL-S languages. Languages for the agreement context include ebXML's Trading-Partner Agreement (TPA) and Collaboration-Protocol Agreement (CPA), WS-Agreement, and the Globus Toolkit's SLAs (Service Level Agreements). The way providers create context specifications is out of the scope of the matching framework. They may use appropriate tools for that purpose. For example, a WSDL editor (e.g., Eclipse WSDL Editor, Cape Clear WSDL Editor, XMLSpy WSDL editor) may be used to create a functional context specification of type WSDL.

A service provider may create one or more context specifications for a given context category. For example, she/he may define WSDL-S and OWL-S functional specifications for a given Web service. These specifications are exposed as (context) *policies* at the *envelope* level. A policy may refer not only to privacy and security features but also other service capabilities such as functional, non-functional (QoS), value-added, and domain capabilities. Each Web service has at most one context policy for each category; this policy envelops all context specifications corresponding to that category. We use the emerging WS-Policy standard for the definition of context policies [22].

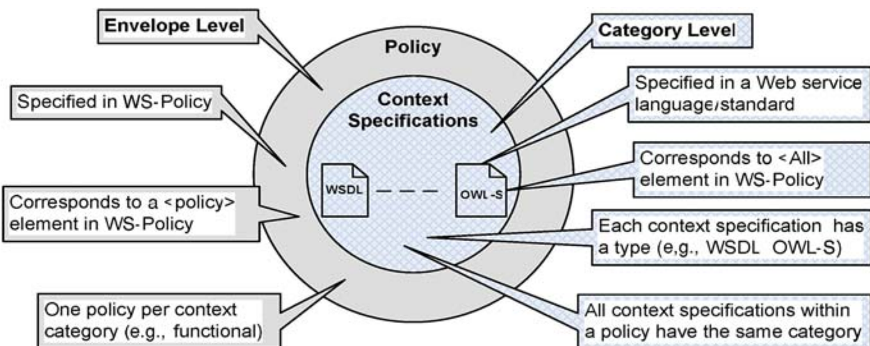

Fig. 4 Two-level approach for modeling Web service contexts

The use of the envelope level enables the wrapping of context specifications in a uniform way. At a high level, all context specifications are seen as described in the same language (i.e., WS-Policy).

WS-Policy provides a general purpose model and syntax to describe and communicate the policies of a Web service. It defines a policy as a collection of *alternatives*, where each policy alternative is a collection of *assertions*. A policy assertion represents a requirement, capability, or other property of a policy subject (e.g., Web service). Assertions indicate domain-specific (e.g., security, privacy) semantics and are expected to be defined in separate, domain-specific specifications. An assertion may have an arbitrary number of child assertions. A policy is described in XML as a policy *expression* through a number of constructs such as "Policy" tag to start and end a policy, "ExactlyOne" tag to contain a collection of alternatives, and "All" tag to include all assertions of an alternative. To illustrate the way context policies are specified, let us consider the following policy expression:

```
<wsp:Policy Name="mypolicy1">
   <wsp:ExactlyOne>
      <wsp:All>
          <context.ws:contextSpecification Type="WSDL-S">
             <!-- link to WSDL-S specification -->
          </context.ws:contextSpecification>
      </wsp:All>
      <wsp:All>
          <context.ws:contextSpecification Type="OWL-S">
             <!-- link to OWL-S specification -->
          </context.ws:contextSpecification>
      </wsp:All>
   </wsp:ExactlyOne>
<wsp:Policy>
```

The example above defines a policy for a functional context category. The way we associate a category to a policy expression is illustrated in Section 3.4. The "wsp" prefix refers to the WS-Policy XML namespace. The policy contains two alternatives, each alternative models a context specification. A context policy contains as many alternatives as the number of functional context specifications defined for the Web service. An alternative includes a "contextSpecification" assertion defined within our "context.ws" namespace. Each assertion has an attribute "Type". A policy document does not contain more than one assertion of a certain type. For example, a Web service does not have more than one WSDL-S specification. Context specifications are given in separate XML documents. The URIs (Uniform Resource Identifier) of those documents (WSDL-S and OWL-S in the example) are given in the "contextSpecification" assertion.

3.4 Context communities

One or more context policies are associated to a Web service. Each policy is defined by the service provider and belongs to a certain category. We cluster context policies into *communities*; a *community* is a "container" that clumps together context policies

related to the same category. A community is itself a Web service that is created and invoked similarly to “regular” Web services[1]. By defining communities as Web services, new context categories may be easily added to the framework. Introducing a new category is done by creating and deploying a new community service corresponding to that category. A community provider may, for example, be a grid service network administrator if the matching framework is to be used in grid environments. Another example of community provider is a domain-specific consortium such as a group of bioinformatics institutions contributing to the development of an experiment specification language. Details about creating communities are given in Section 4.3.

A community service oversees all context policies defined under a given category. It maintains two attributes: *category* and *members*. The *category* of a community is similar to category of its underlying context policies. The *members* attribute contains the list of context policies that fall into the community category. Each community member is defined by the couple (serviceID,policyID) where serviceID is a unique ID of a Web service (e.g., serviceKey in UDDI) that has the policy identified by policyID. As any “regular” Web service, the community service is accessed via operations. Details about community service operations are given in Section 4.2.

Each context policy file is attached to the *registration entry* of corresponding Web service. Registration allows providers to advertise general information about their Web services. This information is used by clients for discovering providers and Web services of interest. UDDI and ebXML Registry are examples of protocols that can be used for the registration of Web services [1]. In this paper, we use UDDI as a registration repository. We adopt the mechanism introduced in WS-PolicyAttachment specification to attach policy expressions to UDDI [23]. We register context policies of a Web service as distinct tModels and then reference these tModels in the businessService entry defined for that service. tModel is a UDDI concept introduced for defining technical fingerprints (or specifications) of Web services [50]. We give below an example of tModel snippet for the previous policy expression:

```
<tModel tModelKey="uuid:...">
   ...
  <categoryBag>
     <keyedReference
       keyName="Reusable policy Expression"
       keyValue="policy"
       tModelKey="uuid:fa1d77dc-edf0-3a84-a99a-5972e434e993"/>
     <keyedReference
       keyName="Policy Expression for mypolicy1"
       keyValue="http://www.example.com/myservice/mypolicy1"
       tModelKey="uuid:a27078e4-fd38-320a-806f-6749e84f8005"/>
     <keyedReference
       tModelKey="uddi:uddi.org:categorization:general_keywords"
       keyName="context.ws:categorization"
       keyValue="functional"/>
  </categoryBag>
</tModel>
```

[1] We use the terms community and community service interchangeably.

The tModelKey attribute in the tModel tag refers to a unique UUID (Universal Unique Identifier) generated by UDDI to refer to the tModel. UUIDs are 128-bit numbers used to uniquely identify an object or entity on the Internet. This UUID also represents the policyID of the context policy modeled by the tModel. The first keyedReference states that the tModel represents a policy expression by using the WS-Policy's built-in category "policy", which is its single valid value. This is necessary to enable UDDI inquiries for policy expressions in general. The second keyedReference designates the policy expression represented by the tModel. The tModelKey must match the fixed tModelKey from the Remote Policy Reference category system (i.e., "uuid:a27078e4-fd38-320a-806f-6749e84f8005"). The keyValue must be the URI that identifies the policy expression (i.e., "http://www.example.com/myservice/mypolicy1"). This keyedReference is necessary to enable UDDI inquiries for particular policy expressions based on their URI. The third keyedReference is a general keyword category. It consists of a namespace identifier (i.e., "context.ws:categorization") that refers to our context category, and a value within this category (i.e., "functional"). This keyedReference is necessary to enable UDDI inquiries for policy expressions based on their categories.

We define two techniques for updating the members attribute of a community service: *pull-based* and *push-based*. In the pull-based technique, the community service (of category *C*) periodically queries the UDDI registry to retrieve the list of context policies of category *C* and their Web service IDs. This is done through the `find_tModel` and `find_service` UDDI inquiries [1, 50]. The main issue of this technique is the frequency for refreshing the list of community members. A high frequency incurs a higher overhead for querying the UDDI registry. A low frequency has the disadvantage of using an obsolete list of community members. In the push-based technique, the members attribute of a community service is updated each time a new context policy with a category *C* is created. In this paper, we adopt the push-based technique. Details about this technique are given in Section 4.2.

3.5 Context policy assistants

Service providers create context policies via *context policy assistants* (CPAs). Each service provider has a CPA attached to it. CPAs are *software agents* that facilitate interaction between (i) providers and the service registry (to store the context policies), and (ii) providers and community services (to update the list of community members). A *software agent* is a piece of software that autonomously acts to carry out tasks on behalf of users [52].

Service providers get their CPAs from a *CPA agent pool*. CPAs are mobile and hence have the capacity to migrate from the pool to service providers's sites. A service provider may create several context specifications (e.g., WSDL-S and WS-SecurityPolicy specifications) using context-specific languages such as those mentioned in Table 1. He/she first gives the URI and category of each context specification to his/her CPA. The CPA creates context policy documents (one document per category) as explained in Section 3.3. Then, it exposes each context policy as a tModel and attaches it to the Web service in the registry as explained in Section 3.4. Finally, the CPA notifies each community of category *C* about the creation of a new context policy with similar category. Each CPA has a list of available communities. An entry

in the list contains the community ID (*C-ID*) and category. Details about the way this list is created and updated are given in Section 4.1.

4 Matching Web service contexts

In this section, we describe our context-based matching framework (Fig. 5). Assume that the composition engine wants to replace a participant *WS* by another Web service. The composition engine sends a matching request `mrq = match-context(WS-ID)` to the Context-based Matching Engine (CME), where WS-ID is the (unique) service ID of *WS* (step 1). CME first retrieves *WS* context policies by submitting appropriate `find_tModel` and `find_service` inquiries to UDDI (step 2). Then, it decomposes the matching request `mrq` into sub-requests `sub-mrq(C,P)` based on the category *C* of each context policy *P* of *WS*. Each sub-request `sub-mrq(C,P)` is forwarded to the participating community service of category *C* (step 3). After processing its sub-request, the community service returns the list of candidate Web services with a context policy that matches *P* (step 4). CME finally gathers the results returned by all participating communities, determines the list of matched Web services, and returns it to the composition engine (step 5).

4.1 The context matching engine

The Context Matching Engine (CME) is the cornerstone of the proposed framework. It receives matching requests from the composition engine and returns a list of *matched* Web services. CME is itself a Web service accessible through a set of operations.

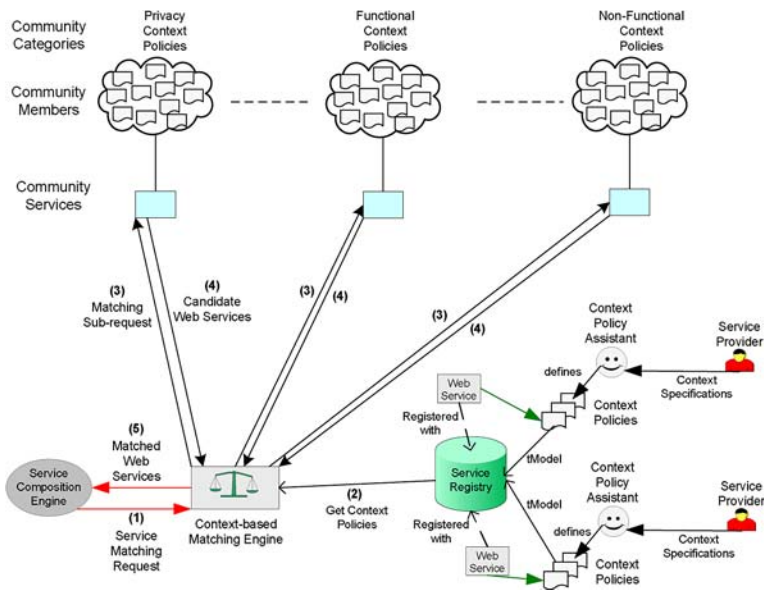

Fig. 5 Overview of the matching framework

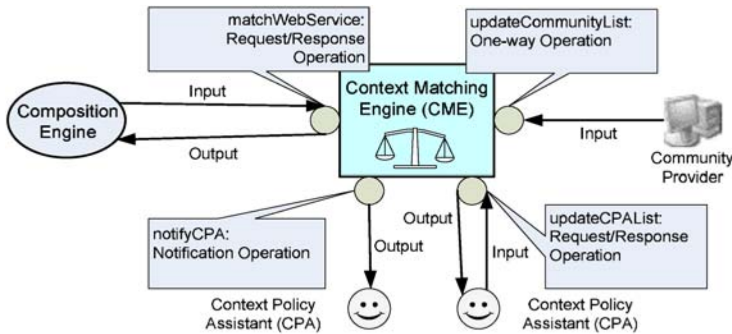

Fig. 6 CME interface

Defining CME as a Web service has three main advantages. First, CME can be integrated into existing service composition engines (e.g., BPEL engine) with minimal changes to the business logic and code of that composition engine. CME promotes significant decoupling with composition engines. It advertises an XML-based interface that can be used to invoke it. This reduces complexity, as composition engine designers do not have to worry about CME implementation details. Second, CME invocation may be included as part of composite service specifications. Composite services involve the orchestration of multiple participant Web services. Composers may specify the invocation of CME matching operation as part of their exception handling policy (e.g., in BPEL). Such exception handling policy may also be automatically inserted by the composite engine. CME will then be seen as any participant in a composite service specification. Third, defining CME as a Web service concords with the OGSA (Open Grid Services Architecture) approach where all grid resources (both logical and physical) are modeled as Web services [16]. CME would be a Web service in the second layer (Web services layer) of OGSA architecture.

CME Web service has four operations (Fig. 6): `updateCommunityList`, `updateCPAList`, `notifyCPA`, and `matchWebService`. The first operation is invoked by community providers at the community creation time. The aim of this operation is to update CME's list of available communities. Each entry in *communityList* contains the community service ID (*C-ID*) and the community category. Each time a new CPA agent is deployed at a service provider's site (from the CPA agent pool), it invokes the `updateCPAList` of CME engine. CME maintains a list of all existing CPAs in CPAList. The CPA sends directions (e.g., URI of a CPA local database) about the way CME can notify it about important events (e.g., availability of a new community). CME returns as output the current *communityList*. This list is used subsequently by the CPA to register its context policies with communities. Whenever a community provider invokes the `updateCommunityList` operation (to report the creation of a new community), CME automatically sends notifications to existing CPAs via the *notifyCPA* operation. Each notification contains the newly created community service ID (*C-ID*) and its category.

The `matchWebService` operation returns a list of Web services (called *matched* Web services) from the registry whose context policies match the context policies of a given Web service (called *source* service). The composition engine invokes this

```
(00)  matchWebService Algorithm  {   /* Executed by CME Engine */
(01)  /* Phase 1 of the Algorithm:  Polling Phase */
(02)    replies = 1;
(03)    Upon Invocation Of matchWebService (WS-ID) Do
(04)          all-policies = getAllPolicy (registry, WS-ID);
(05)          For Each policyID in all-policies Do
(06)              policyCategory = getCategory (policyID);
(07)              C-ID = getCommunityID (policyCategory, comunityList);
(08)              invoke matchContextPolicy (policyID, WS-ID) Of C-ID;
(09)          End Do
(10)    End Do
(11)  /* Phase 2 of the Algorithm:  Decision Phase */
(12)    Upon Reception Of  (WS-ID, candidateWS) From C-ID Do
(13)          If replies == 1
(14)                  Then matchedWS = candidateWS
(15)                  Else matchedWS = matchedWS ∩ candidateWS;
(16)          If replies == |all-policies| Then Return (WS-ID, matchedWS);
(17)                                       Else replies = replies + 1;
(18)    End Do
(19)  }
```

Fig. 7 Two-phase matching algorithm

operation by sending the unique ID of the source service. It is up to the composition engine's internal business logic to select one service from the list of matched Web services returned by CME. For example, a composition engine may select the Web service with the "best" quality of service. The selection process is out of the scope of the matching framework.

Figure 7 gives the algorithm executed by CME's `matchWebService` operation. The algorithm is composed of two phases: the *polling* phase and *decision* phase. In the polling phase (lines 1–10), CME sends matching requests to relevant community services. CME first gets the IDs of all context policies attached to the source service WS-ID (line 4). For that purpose, it submits a `find_tModel` inquiry to the UDDI registry. The inquiry uses as argument the tModelKey defined for WS-Policy expressions (see Section 3.4). It returns a list of policyIDs of the corresponding context policies. A `get_tModelDetail` inquiry is also executed to get the tModel policy document of each policyID. CME then determines the policyCategory of each tModel policy document (lines 5–6). To this end, it checks the general keyword category represented by the third keydReference in the tModel (see Section 3.4). CME gets the C-ID of the community that has policyCategory as a category by accessing the communityList table (line 7). It finally invokes the `matchContextPolicy` offered by the community service identified by C-ID (line 8). As the size of a policy document may be large (e.g., WSDL-S document with several XML elements), only policyID and WS-ID are forwarded to the community. Communities may retrieve context information via the registry. The previous steps (lines 6–8) are repeated for each context policy attached to the source service. The way community services process `matchContextPolicy` requests is community-dependent and done transparently to CME. Details about the `matchContextPolicy` operation are given in Section 4.2.

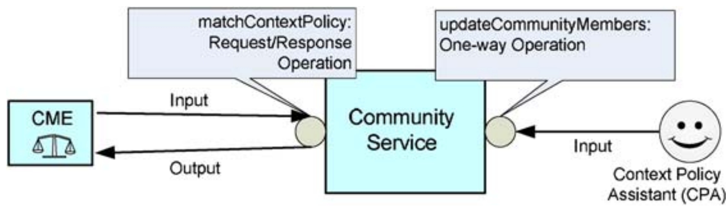

Fig. 8 Community service interface

In the second phase (decision phase), CME consolidates the results received from communities involved in the polling phase (lines 11–19). Each community service C-ID replies with a candidateWS list. The list includes the IDs of services whose context policy matches (as defined by the community) the context policy (with similar category) of the source service. We use a variable "replies" which contains the number of votes received so far by CME. The variable is initialized and updated in lines 2 and 18 respectively. The decision phase terminates when CME receives the votes of all communities participating in the first phase (i.e., $replies = |all\text{-}policies|$). In this case, CME returns the list matchedWS to the composition engine (line 16). A Web service belongs to the matchedWS list if each context policy in the source service is matched by a corresponding context policy in that service. This is done by computing the intersection of candidateWS lists returned by all community services involved in the polling phase (lines 13–15).

4.2 Inside view of a community service

Community services provide a peer-to-peer topology for matching context policies. Each community handles context policies that belong to a specific category (i.e., the community's category). The interface exposed by a community service is composed of two operations (Fig. 8): `updateCommunityMembers` and `matchContextPolicy`. The `updateCommunityMembers` operation enables interactions between communities and CPAs (context policy assistants). As described in Section 3.5, service providers create context policies via their CPAs. At policy definition time, the CPA communicates with the corresponding community to update its members list. If a policy (identified by P-ID) of category *C* is created for a Web service WS-ID, then CPA invokes the `updateCommunityMembers` operation of the community of category *C*. This operation is one-way (i.e., has only an input message) and contains WS-ID and P-ID as input parameters.

The `matchContextPolicy` operation allows CME to send context matching requests to community services (Fig. 7, line 8). It is an input/output operation; the input message is composed of the WS-ID of the source Web service (i.e, service to be matched) and a policyID of that service policy (called *source* policy). The operation compares the source policy with all policies (called *member* policies) that are members of the community. The output message contains the IDs of *candidate* Web services; a candidate is a Web service with a policy (called *candidate* policy) that matches the source policy.

The `matchContextPolicy` operation relies on *context rules* to decide whether a member policy matches a source policy. Each community has its own set of context rules stored in a *context rule base* (CxRB). The syntax of a context rule is given below:

Context Rule *rule-name*
  Context Property *property-name*
  Instances *source-instance, member-instance*
  Type *context-specification-type*
  Action *matchContextProperty(source-instance, member-instance)*

A context rule is identified by a name and corresponds to a specific *context property*. A *context property* is a matching criterion used by the community to decide whether a member policy matches the source policy. Several context properties may be associated to a given context category. For example, a non-functional context policy includes a set of quality of service parameters (e.g., response time) associated to the service. Each qualitative parameter is seen by the non-functional community service as a context property. This community has as many context rules as the number of qualitative parameters (e.g., cost, availability) defined in non-functional contexts. A property has a *source* and *member instance*; a source (member) instance is the value of the property in the source (member) policy. For example, the actual response time of the `homology search` Web service is the source instance of the *response time* property. As explained in Section 3.3, each policy may contain several context specifications of different types (e.g., WSDL-S, DAML-S, OWL-S). A rule deals with a context property as defined in a context specification type. This information is specified in the *type* clause.

The *action* clause contains a boolean function *matchContextProperty(source-instance,member-instance)* which returns *true* iff the member instance matches the source instance. The internal business logic of this function is property-dependent. It is up to the community provider to identify context properties and figure out the appropriate code associated to each property. Community providers may include the code of *matchContextProperty()* as part of the rule specification; they may also define that code as a stored procedure and provide a link to it within the rule specification. Below is an example of rule created for the *response time* property of a non-functional community:

Context Rule *rule-RT*
  Context Property *response time*
  Instances $time_S, time_M$
  Type HQML
  Action *matchContextProperty*($time_S, time_M$)
    { **If** $time_M \leq time_S$ **Then return** *true* **Else return** *false* }

The rule states that a member's response time matches a source's response time if the member's is at most equal to the source's (i.e., the response time of the member Web service is shorter or the same as the response time of the source). As specified in the *Type* clause, the rule compares *response time* property instances of HQML (Hierarchical QoS Markup Language) context specifications [19].

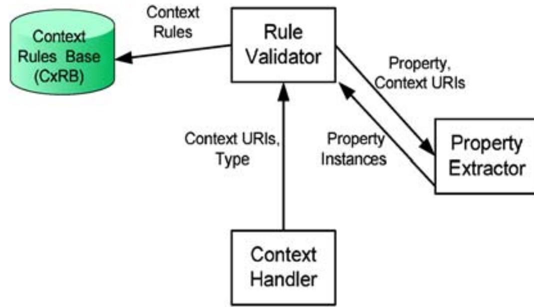

Fig. 9 Components of the matchContextPolicy operation

The `matchContextPolicy` operation uses three major components to determine candidate services (Fig. 9): *context handler*, *rule validator*, and *property extractor*. The algorithm executed by the `matchContextPolicy` operation of a community C_i is summarized in Fig. 10. Method calls that end with CH, RV, and PE are executed by the context handler, rule validator, and property extractor respectively. The context handler first retrieves the source policy document P_S (line 3). For each context specification C_S (modeled as an alternative in the policy document), it gets its type and URI (line 4–6). The context handler looks-up in C_i's members list for each policy P_M that includes a context specification C_M of type similar to type-C_S and gets its URI (line 8–10). The *getURL-CH* (P_M, type-C_S) call return null if there is no context specification of type equal to type-C_S in P_M (line 11). If a service member already belongs to the list of candidate Web services, then it is skipped (line 8). There is no need to process the policy document of a member service if that service has already been selected by C_i as a candidate.

As mentioned in Section 3.3, P_M contains at most one specification of type equal to type-s. P_M matches P_S if the following predicate is true: $\exists\, C_S \in P_S\ \exists\, C_M \in P_M \mid C_M$ *matches* C_S. P_M matches P_S if there is a context specification in P_M that *matches* a context specification in P_S. Assume that a source service W_S has three context specifications WSDL-S, DAML-S, and OWL-S defined within its functional context policy P_S. The three specifications are alternatives advertised by W_S provider to describe its functional context. Clients (e.g. composition engine) may consider any of these alternatives as a functional specification of W_S. Hence, a service with policy P_M should have one specification that matches the WSDL-S *or* DAML-S *or* OWL-S specification of W_S. There is no need to find a match for the three specifications at the same time.

To check whether C_M *matches* C_S, the context handler forwards each tuple (P_S, uri-C_S, P_M, uri-C_M, type-C_S) to the rule validator. The rule validator retrieves from CxRB rule base the set $\mathcal{R}$ of all rules of type equal to type-C_S (line 7). The rule validator then gets the property name P_j of each rule R_j in $\mathcal{R}$ and passes P_i along with the URIs of C_S and C_M to the property extractor (lines 14–16). The property extractor parses the XML documents of C_S and C_M and returns the value of P_i in each document. Let I_S and I_M be the source's and member's instance respectively. The rule validator checks whether I_M matches I_S as defined in the *matchContextProperty* function of R_i (line 17). C_M matches C_S if the *matchContextProperty* function of each rule in $\mathcal{R}$ returns the boolean true (line 18). In this case, the WS-ID of the member service

```
(00)  matchContextPolicy (P_S-ID, source) Algorithm {
(01)    /* Executed by Community C_i */
(02)    candidateWS = ∅;
(03)    P_S = getPolicy-CH (registry, P_S-ID);
(04)    For each C_S ∈ P_S Do
(05)     type-C_S = getType-CH (P_S, C_S);
(06)     uri-C_S = getURL-CH (P_S, type-C_S);
(07)     R = getRules-RV (CxRB, type-C_S);
(08)          For Each (WS-ID, policyID) ∈ C_i.members | WS-ID ∉ candidateWS Do
(09)           P_M = getPolicy-CH (registry, policyID);
(10)           uri-C_M = getURL-CH (P_M, type-C_S);
(11)           If uri-C_M == null Then Continue;
(12)           match = true;
(13)                For Each rule R_j ∈ R Do
(14)                 P_j = getProperty-RV (R_j);
(15)                 I_S = extractInstance-PE (P_j, uri-C_S);
(16)                 I_M = extractInstance-PE (P_j, uri-C_M);
(17)                 match = R_j.matchContextProperty-RV (I_S, I_M);
(18)                 If match == false Then Break;
(19)                End Do
(20)           If match Then candidateWS = candidateWS ∪ {WS-ID};
(21)          End Do
(22)    End Do
(23)    Return (source, candidateWS);
(24)  }
```

Fig. 10 Matching contexts in communities

corresponding to P_M is added to the list of candidate Web services (line 20). Finally, the `matchContextPolicy` operation returns the source ID and list of candidates as a result (line 23).

The context matching algorithm depicted in Fig. 10 compares context specifications that have the same type (see lines 6 and 10). Comparing context specifications that have different types within a community (e.g., DAML-S with WSDL-S) requires dealing with the issue of defining mappings between disparate languages or ontologies. This issue is out of the scope of this paper. However, our framework can be extended to deal with ontology mapping. Techniques such as the ones described in [14] may be adopted.

4.3 Community factory

Web services operate in a highly dynamic environment where changes can occur to adapt to actual business climate (e.g., politic, economic, organizational). For example, new government regulations may require each Web service in the healthcare domain to expose its privacy practices. Generally speaking, changes are initiated by Web service providers or community providers. At the Web service side, service providers may create a new context policy document of category *C*. For that purpose, the service provider's CPA invokes the `updateCommunityMembers` operation of the community of category *C* (see Section 4.2).

At the community side, community providers may create community services at any time. This happens when a new context category or subcategory is added to the context categorization ontology (Fig. 3). Assume for example that a *negotiation* context is added to the set of value-added contexts. A community provider (e.g., consortium of companies developing a negotiation standard for Web services) would like then to create a new community for this category. We define a *design pattern* for the creation of community services. Design patterns are a powerful tool for designing flexible software [17]. They provide well-tested solutions to recurring software design problems. One such widely used pattern is the *Abstract Factory* pattern. This pattern provides an interface for creating families of related or dependent classes without having to specify the actual classes. The Abstract Factory uses *Factory Methods* to handle the actual creation of specific objects [17]. This is called a factory pattern because it is responsible for "manufacturing" objects.

We define an abstract factory design pattern for communities called *Community Abstract Factory*. This pattern's interface includes four methods: `matchContextPolicy`, `notifyCME`, `extractInstance`, and `createCxRB`. Community providers create concrete implementations of the abstract factory. Each concrete implementation represents a community service. The `matchContextPolicy` method enables communities to compare context policies as described in Fig. 10. The `notifyCME` method allows community providers to notify CME engine whenever they create new communities. For that purpose, it invokes the `updateCommunityList` operation exposed by CME engine (Fig. 6). The `extractInstance` method is used by a community's property extractor to retrieve the instance of a context property from a context specification corresponding to that community. The implementation of this method depends on the context category language. It is up to community providers to implement this method as needed. For example, the provider of a functional community should specify in `extractInstance` the way each qualitative parameter is retrieved from an HQML specification [19]. The `createCxRB` allows community providers to create CxRB rule bases and populate them with appropriate context rules. Community providers should identify the context property of each rule. They also need to specify the business logic of the *matchContextProperty()* function.

5 Implementation

Figure 11 gives the implementation architecture of the proposed matching framework. We implement CME and communities as Java Web services using Eclipse IDE. Details about the way community services are implemented are given in Section 5.1. We use Apache Axis 1.2.1 to deploy and invoke those Web services. To actually host CME and community services, we use Apache Tomcat server (version 5.X). We use jUDDI, an open source Java implementation of UDDI, as a service registry. Each community implements the community factory pattern. Service providers are represented by their CPAs. Each service provider downloads its CPA from a CPA agents pool using a CPA editor (implemented in JSP 1.0 and Java Servlets 2.1). CPAs are mobile agents developed in IBM's Aglets Software Development Kit (version 2).

We provide two mechanisms for submitting matching requests to CME engine. In the first mechanism, users enter a matching request via a graphical interface (imple-

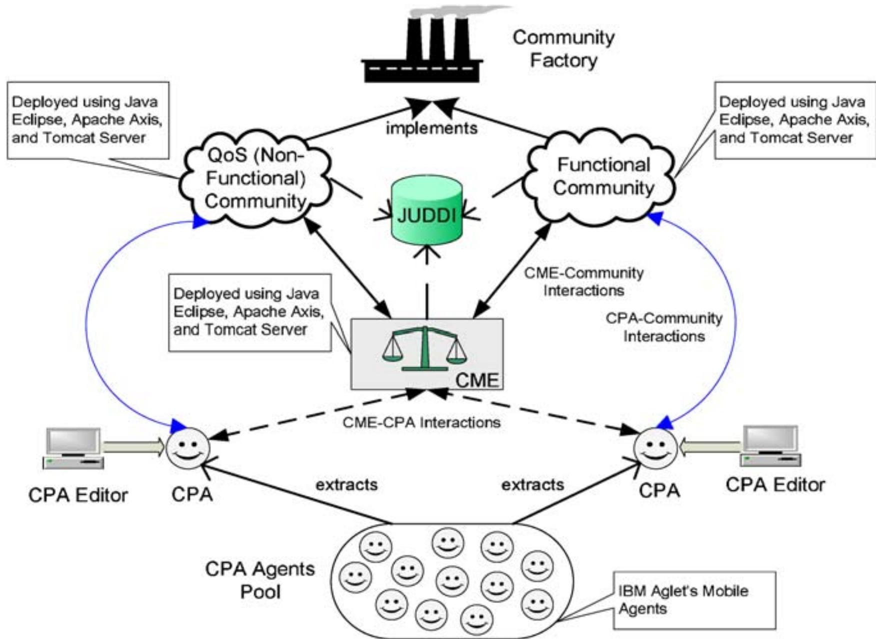

Fig. 11 Implementation architecture

mented in HTML/servlets). The specified matching request includes context specifications along with their types. At the server side, we automatically generate policies based on context specifications, create a new service entry in the jUDDI registry, and invoke CME's *matchWebService()* for the newly created service. This mechanism gives users control over the cases that need to be tested in the matching framework (e.g., test the framework for a specific context category, type, and property instances). In the second mechanism, a Web service randomly invokes CME's *matchWebService()* operation for randomly selected source services. These services are pre-defined and pre-registered in the jUDDI registry. This mechanism simulates matching requests submitted from a composition engine. It is suitable for conducting experiments to assess the performance of the proposed framework since a large number of random matching requests may be generated.

5.1 Implementing community services

We are currently implementing two context categories: QoS (non-functional category) and functional. Each category corresponds to a community service. For QoS, we are adopting the QoS model presented in [54]. We are using WSDL as a type for the functional category. Other context categories and types will be added in the future. To illustrate the way community service are implemented, let us consider the QoS community. In the current implementation, a QoS policy is composed of one context specification that includes the following attributes: price, duration, reputation, and availability [54]. The price of a service is the fee that a client has to pay for invoking that service. The duration measures the expected delay in seconds between the moment

Table 2 Context Rules implemented in the QoS community

Context Rule *Price*	**Context Rule** *Duration*
Context Property *price*	**Context Property** *duration*
Instances $Price_S$, $Price_M$	**Instances** Dur_S, Dur_M
Type QoS	**Type** QoS
Action *matchContextProperty*($Price_S$,$Price_M$)	**Action** *matchContextProperty*(Dur_S,Dur_M)
{ **If** $Price_M \leq Price_S$	{ **If** $Dur_M \leq Dur_S$
Then return *true* **Else return** *false* }	**Then return** *true* **Else return** *false* }
Context Rule *Reputation*	**Context Rule** *Availability*
Context Property *reputation*	**Context Property** *availability*
Instances Rep_S, Rep_M	**Instances** $Avail_S$, $Avail_M$
Type QoS	**Type** QoS
Action *matchContextProperty*(Rep_S,Rep_M)	**Action** *matchContextProperty*($Avail_S$,$Avail_M$)
{ **If** $Rep_M \geq Rep_S$	{ **If** $Avail_M \geq Avail_S$
Then return *true* **Else return** *false* }	**Then return** *true* **Else return** *false* }

when a request is sent to a service and the moment when the results are received. The reputation of a service *s* is a measure of its trustworthiness. It mainly depends on end user's experiences of using the service. Different end users may have different opinions on the same service. The value of the reputation is defined as the average ranking given to the service by end users. The availability of a service *s* is the probability that the service is accessible. Each QoS attribute corresponds to a context property. Hence, we have a total of four context rules (one for each context property). These rules are summarized in Table 2. Let us consider two services WS_{source} and WS_{member}. The *price* rule states that WS_{member} matches WS_{source} if WS_{member} has a price that is at most equal to WS_{source}'s price. The *duration* rules indicates that WS_{member} matches WS_{source} if WS_{member}'s execution does not last more than WS_{source}'s execution. The *reputation* rule specifies that WS_{member} matches WS_{source} if WS_{source} is not more reputable than WS_{member}. The *availability* rule states that WS_{member} matches WS_{source} if WS_{member} is at least as available as WS_{source}.

The context rules specified in Table 2 are stored in the community's CxRB rule base. CxRB is implemented as a relational table in Oracle9i. CxRB table has three attributes: rule name, property name, and type. Each context rule will be represented by a tuple in the table. The *matchContextProperty()* function of each rule is saved in Oracle9i as a Java stored procedure. The source and member instances are not saved in CxRB table since their values depend on the source service (different from a matching request to another) and the current member service being processed. The source and member instances are passed as parameters to stored procedures by the community's *matchContextPolicy()* algorithm (Fig. 10). Community providers populate the CxRB table and save the stored procedures at community creation time. This is done as part of the `createCxRB` method defined in the community abstract factory (Section 4.3). The use of a relational table and stored procedures concept for CxRB has two major advantages. First, the *matchContextPolicy()* algorithm (Fig. 10) may use simple SQL queries to fetch appropriate context rules. One such SQL query is the one that returns the rule names that correspond to a specific property name and type. Second, community providers may modify CxRB at any time without changing

Table 3 Symbols and Parameters

Variables	
N_{com}	Number of communities
N_p	Number of policies per service
N_{CS}	Number of context specifications per policy
N_{memb}	Number of members per community
N_{R-t}	Number of rules per type t
N_S	Number of services in the registry
Performance measurement parameters	
T_{P-all}	Time to fetch all policies of a service from the registry
T_{P-one}	Time to fetch one policy of a service from the registry
T_{CS}	Time to get a context specification
T_{XML}	Time to parse a service description
T_{Net}	Network transmission delay
T_{CxRB}	Time to fetch rules from CxRB rule base
T_{MCR}	Time to execute the matchContextProperty() function within a rule
T_{rep}	Time spent by CME to assess a reply from a community

the community logic and code. For example, they may add new rules, delete rules, modify existing rules, and modify/add stored procedures.

5.2 Performance analysis

In this section, we study the performance of the proposed framework for context-based matching. For that purpose, we define an analytical model and analyze the performance of our framework using that model. Performance study via simulation or real-world experiments will be reported in a future work. Table 3 defines the parameters and symbols used in the analytical model. We compute the average matching time T_{match}. Thus, T_{match} is equal to $(T_{match}^{min} + T_{match}^{max})/2$ where T_{match}^{min} and T_{match}^{max} are the minimum and maximum matching times respectively.

As specified in Fig. 7, the global matching time T_{match} includes a polling time T_{poll} and decision time T_{dec} ($T_{match} = T_{poll}+T_{dec}$). T_{poll} is composed of the time T_{poll1} spent by CME to decompose requests and invoke community services and the time T_{poll2} spent by each involved community to process the its sub-request. In the best case scenario (i.e., minimum time), the source service *WS* has only one policy. Hence, $T_{poll}^{min} = T_{poll1}^{min} + T_{poll2}^{min}$. In the worst case scenario (i.e., maximum time), the source service has N_{com} policies and all sub-requests are processed sequentially. Hence, $T_{poll}^{max} = T_{poll1}^{max} + N_{com} \times T_{poll2}^{max}$.

The following time delays have to be considered for computing T_{poll1}: (1) Time spent by CME to retrieve the context policies of WS (Fig. 7, line 4); (2) Time spent by CME to decompose the matching request into sub-requests (Fig. 7, lines 5–7); and (3) Time to send each sub-request to the community service of the related category (Fig. 7, lines 8). T_{poll1}^{min} and T_{poll1}^{max} correspond to the cases where WS has one and N_{com} policies, respectively. Hence, $T_{poll1}^{min} = T_{P-one} + T_{XML} + T_{Net}$ and $T_{poll1}^{max} = T_{P-all} + N_{com} \times (T_{XML} + T_{Net})$. Before computing T_{poll2}, let us first determine T_{dec} as part of the algorithm presented in Fig. 7. In the best case scenario, $T_{dec}^{min} = T_{Net} + T_{rep}$ since WS has only one policy and therefore CME receives one reply from a community. In the worst case scenario, $T_{dec}^{max} = N_{com} \times (T_{Net} + T_{rep})$ since CME receives sequential

replies from all community services. We summarize below the formulas for T^{min}_{poll}, T^{max}_{poll}, T^{min}_{poll1}, T^{max}_{poll1}, T^{min}_{dec}, and T^{max}_{dec}:

$$
\begin{aligned}
T^{min}_{poll} &= T^{min}_{poll1} + T^{min}_{poll2}\\
T^{max}_{poll} &= T^{max}_{poll1} + N_{com} \times T^{max}_{poll2}\\
T^{min}_{poll1} &= T_{P-one} + T_{XML} + T_{Net}\\
T^{max}_{poll1} &= T_{P-all} + N_{com} \times (T_{XML} + T_{Net})\\
T^{min}_{dec} &= T_{Net} + T_{rep}\\
T^{max}_{dec} &= N_{com} \times (T_{Net} + T_{rep})
\end{aligned}
$$

Let us now compute T_{poll2} based on the algorithm given in Fig. 10. The community first retrieves WS policy from the registry in a T_{P-one} time delay (Fig. 10, line 3). For each context specification (for a total of N_{CS}), it parses the policy in T_{XML} time delay to get the type and URI of the specification and retrieves all matching rules (in a T_{CxRB} time delay) corresponding to the same type from CxRB rule base (Fig. 10, line 4–7). Then, the community browses all members (for a total of N_{memb}). For each member, it fetches its policy from the registry and parses it to get the URI of a relevant context specification in a $T_{P-one} + T_{XML}$ time delay (Fig. 10, line 8–12). Next, the community retrieves the source and member specifications based on the URIs (in $2 \times T_{CS}$ time delay). Finally, for each matching rule (for a total of N_{R-t}), it parses the source and member specifications to extract the source and member property instances in a $2 \times T_{XML}$ time delay, and executes the *matchContextProperty()* function within that rule in a T_{MCR} time delay (Fig. 10, line 14-18). We can now determine T^{min}_{poll2} and T^{max}_{poll2}. T^{min}_{poll2} corresponds to the case where $N_{CS} = 1$, $N_{memb} = 1$, and $N_{R-t} = 1$. T^{min}_{poll2} corresponds to the case where $N_{memb} = N_S$ (all services are members of the community). The formulas for T_{poll2}, T^{min}_{poll2}, and T^{max}_{poll2} are given below:

$$
\begin{aligned}
T_{poll2} &= T_{P-one} + N_{CS} \times (T_{XML} + T_{CxRB} + N_{memb}\\
&\qquad \times (T_{P-one} + T_{XML} + 2 \times T_{CS} + N_{R-t} \times (2 \times T_{XML} + T_{MCR})))\\
T^{min}_{poll2} &= 2 \times T_{P-one} + T_{CxRB} + 2 \times T_{CS} + 4 \times T_{XML} + T_{MCR}\\
T^{max}_{poll2} &= T_{P-one} + N_{CS} \times (T_{XML} + T_{CxRB} + N_S\\
&\qquad \times (T_{P-one} + T_{XML} + 2 \times T_{CS} + N_{R-t} \times (2 \times T_{XML} + T_{MCR})))
\end{aligned}
$$

Finally, based on our previous analysis, T_{match}, T^{min}_{match}, and T^{max}_{match} are defined as follows:

$$
\begin{aligned}
T^{min}_{match} &= T^{min}_{Poll1} + T^{min}_{Poll2} + T^{min}_{dec}\\
T^{max}_{match} &= T^{max}_{Poll1} + N_{com} \times T^{max}_{Poll2} + T^{max}_{dec}\\
T_{match} &= \frac{1}{2} \times \left(T^{min}_{Poll1} + T^{max}_{Poll1} + T^{min}_{Poll2} + N_{com} \times T^{max}_{Poll2} + T^{min}_{dec} + T^{max}_{dec}\right)
\end{aligned}
$$

6 Related work

In this section, we overview major techniques related to our approach. We first compare our approach to matching techniques for Web services. Then, we position our work with existing context-oriented Web service frameworks. Finally, we discuss related policy-based techniques for Web services.

6.1 Matching techniques for web services

Several techniques have been proposed to deal with Web service matching. LARKS defines five techniques for service matchmaking: context matching, profile comparison, similarity matching, signature matching, and constraint matching [47]. Matching services to requests is performed by using any combination of the above techniques. The ATLAS matchmaker defines two methods for comparing service capabilities described in DAML-S [42, 44]. The first method compares functional attributes to check whether advertisements support the required type of service or deliver sufficient quality of service. The second compares the functional capabilities of Web services in terms of inputs and outputs. [8] presents a service matching technique for pervasive computing environments. Service descriptions are provided in DAML-S. They also include platform specific information such as processor type, speed, and memory availability. The composition manager uses a semantic service discovery mechanism to select participant services. This mechanism is based on DReggie, a Jini-based semantic discovery framework [7]. Li and Harrocks [28] adopts techniques from knowledge representation to match DAML-S service capabilities. In particular, it defines a description logic (DL) reasoner; advertisements and requests are represented in DL notations. Another DAML-S based matchmaker implementation is KarmaSIM [41] where DAML-S descriptions are described in terms of a first-order logic language (predicates) and then converted to Petri-nets where the composition can be simulated, evaluated and performed. Other service matching techniques are also presented in [3, 21, 35]. However, they techniques mostly focus on comparing syntactic attributes of Web services.

We identify several differences between our approach for service matching and the aforementioned frameworks. First, existing frameworks use a limited set of matching attributes. They merely compare text descriptions, signatures (inputs and outputs), and logical constraints about inputs and output. In our approach, we define an ontology-based categorization of contexts that includes functional, non-functional, value-added, and domain context specifications. The matching engine uses those context specifications to decide about the outcome of the matching process. Second, existing frameworks model matching attributes as part of Web service descriptions. In our approach, we define a two-level mechanism for modeling the contextual information to be used in a service matching process. Service descriptions represent part of the context specified in the first level. Other contexts such as security, privacy, agreement, and quality of service may also be specified in that level. We use WS-Policy standard in the second level to provide a uniform (policy-based) envelop for context specifications. Third, we propose a peer-to-peer topology for matching context-oriented services in Web service composition. The topology includes a context matching Web service and

set of specialized community services. The matching process is performed through interactions between matching-related Web services. Finally, our approach is more dynamic than the existing frameworks for Web service matching. New context categories and community services can be added on the fly. Context policies may join and leave community services at any time. Context rules within a community may be updated, removed, and added dynamically.

6.2 Context-oriented Web services

Several context-aware approaches have recently been proposed to enhance Web service discovery and composition mechanisms. Lee and Helal [26] proposes a context-aware service discovery technique for mobile environments. It defines the context of a Web service as a set of attributes included in the service description. Examples of context attributes include user's location and network bandwidth. The discovery engine first lookups for Web services based on traditional criteria (e.g., service category in UDDI). Then, it reduces the qualified services to be returned to clients through context attribute evaluation. This approach uses contextual information for service discovery *not* for service composition. Additionally, it focuses on client-related contextual information. It does not seem to consider provider-related context which is important for Web service composition. Finally, the definition of context in [26] is limited to some attributes added to service descriptions. We adopt a more generic definition of Web service context through an ontology-based categorization of contextual information. Maamar et al. [30] proposes an approach for Web service composition based on the use of agents and context. The suggested context model comprises three types of context: I-context refers to a Web service instance context, W-context is the Web service context that is defined by means of I-contexts, and C-context is the context of the composite service and is defined by the respective W-contexts. Maamar et al. [30] focuses on run-time context which includes information related to the execution of composite Web services and their participants (e.g., number of current service instances and their status). In this paper, we adopt a complementary approach by considering context needed for matching services in Web service composition. We propose a novel categorization of contextual information suitable to Web service matching. Our approach can be combined with the one presented in [30] to consider both run-time and provider/client-related context during service matching.

Contextualization is proposed at the Web service deployment, composition and conciliation or matching levels in [29]. The description of contexts is assumed to occur along three categories: profile, process model, and grounding. The profile describes the arguments and capabilities of context (what does the context require and provide). The process model suggests how context collects raw data from sensors and detects changes that need to be submitted to the Web service. Finally, the grounding defines the bindings (protocol, input/output messages, etc.) that make context accessible to a Web service. The authors did not however mention how relevant contexts are elicited in a service matchmaking process. A policy-based approach for developing context-oriented Web services is introduced in [32]. A Web service policy contains two elements: the resource element which identifies the computing means on which Web services operate, and the user element which identifies the personalization that

Web services are subject to [31]. Context categorization proposed in our approach is more generic than the one defined in [32]. Additionally, [32] does not consider the issue of service matching.

6.3 Policy-based Web services

An approach for matching non-functional properties of Web services represented using WS-Policy is defined in [51]. Our definition of policy is however broader; it encompasses non-functional attributes, as well as other context-related specifications (functional, value-added, and domain). Additionally, we provide an end-to-end framework for processing service matching requests. A policy-based approach for Web service composition is proposed in [9]. The authors propose a classification of rules into syntactic, semantic and policy rules to discover and and compose Web services. The authors claim that the provision of rules with topic concepts allows the system to identify the relevant rules in a certain domain which are going to be used to select appropriate Web services for composition. Unlike our proposed model which is WS-Policy standard compliant, the policy specification used in [9] does not provide a uniform (policy-based) envelop for context specifications. Additionally, our definition of policy is broader, encompassing contextual information such as functional, non-functional, value-added, and domain contexts. We also propose a peer-to-peer framework for matching context policies. A specification of Web service business rules based on WS-Policy framework is suggested in [46]. The policy assertions are modeled by the actions relating to the service and the conditions for performing them. Sriharee et al. [46] focuses on checking the policy of a Web Service against a service consumer's request. It does not provide a matching technique for Web service composition which could exploit such integration of context-related information in the policy assertions.

7 Conclusion

We presented a context-based matching framework for Web service composition. The framework relies on an ontology-based categorization of service contexts. Providers expose their service contexts as policies using context policy assistants (CPAs). CPAs implement a two-level mechanism for modeling Web service contexts. The matching process is performed via peer-to-peer interactions between a context-based matching engine (CME), CPAs, and community services. In this paper, we consider full matching between Web services where each context of a source service should be matched by a policy of the candidate service. As future work, we are investigating techniques for enabling partial service matching. Another possible extension is the use of ontology mapping approaches to support the matching of contexts that have different types within the same category (e.g., WSDL-S vs. DAML-S). Finally, we will conduct experiments to assess the performance of the proposed framework and compare the experiments' results with those obtained in our analytical analysis.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju, Web Services: Concepts, Architecture, and Applications, Springer Verlag (ISBN: 3540440089), June 2003.
2. Y. Atif, "Building trust in E-commerce," IEEE Internet Computing, vol. 6, no. 1, pp. 18–24, 2002.
3. K. Baina, K. Benali, and C. Godart, "A process service model for dynamic enterprise process interconnection," in CoopIS Conf., pp. 239–254, September 2001.
4. R. Bhatti, E. Bertino, and A. Ghafoor, "A trust-based context-aware access control model for Web-services," Distributed and Parallel Databases, vol. 18, no. 1, pp. 83–105, July 2005.
5. P. Brezillon, "Context-based modeling of operators' Practices by Contextual Graphs," in Human Centered Processes: 14th Mini Euro Conference, 2003.
6. F. Casati and M.-C. Shan, "Dynamic and adaptive composition of E-services," Information Systems, vol. 26, no. 3, pp. 143–163, 2001.
7. D. Chakraborty, F. Perich, S. Avancha, and A. Joshi, "DReggie: a smart service discovery technique for E-commerce applications," in Workshop at the 20th Symposium on Reliable Distributed Systems, October 2001.
8. D. Chakraborty, F. Perich, A. Joshi, T. Finin, and Y. Yesha, "A reactive service composition architecture for pervasive computing environments," in 7th Personal Wireless Communications Conference, pp. 53–62, October 2002.
9. S.A. Chun, V. Atluri, and N.R. Adam, "Using semantics for policy-based web service composition," Distributed and Parallel Databases, vol. 18, no. 1, pp. 37–64, 2005.
10. L.F. Cranor, "P3P: making privacy policies more useful," IEEE Security and Privacy, vol. 1, no. 6, pp. 50–55, November 2003.
11. F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web services Web: an introduction to SOAP, WSDL, and UDDI," IEEE Internet Computing, vol. 6, no. 2, pp. 86–93, 2002.
12. G. Della-Libera, M. Gudgin, P. Hallam-Baker, M. Hondo, H. Granqvist, C. Kaler, H. Maruyamaand M. McIntosh, A. Nadalin, N. Nagaratnam, R. Philpott, H. Prafullchandra, J. Shewchuk, D. Walter, and R. Zolfonoon, "Web services security policy language (WS-SecurityPolicy)," http://specs.xmlsoap.org/ws/2005/07/securitypolicy/ws-securitypolicy.pdf, July 2005.
13. A.K. Dey, "Providing architectural support for building context-aware applications," Ph.D. Dissertation, Georgia Tech, December 2000.
14. A. Doan, "Learning to map between structured representations of data," Ph.D. Dissertation, University of Washington, 2002.
15. D. Fensel, Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce, Springer Verlag (ISBN: 3540003029), September 2003.
16. I. Foster and C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, 2nd Edn, Morgan Kaufmann (ISBN: 1-55860-933-4), November 2004.
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley (ISBN: 0201633612), January 1995.
18. A.J. Gonzales and R Ahlers, "Context-based representation of intelligent behavior in training simulations," International Transactions of the Society for Computer Simulation, 1999, pp. 153–166.
19. X. Gu, K. Nahrstedt, W. Yuan, D. Wichadakul, and D. Xu, "An XML-based quality of service enabling language for the web," J. Vis. Lang. Comput., vol. 13, no. 1, pp. 61–95, 2002.
20. J.M. Hancock and M.J. Zvelebil, Dictionary of Bioinformatics and Computational Biology, Wiley-Liss (ISBN: 0471436224), August 2004.
21. J.V.D. Heuvel, J. Yang, and M.P. Papazoglou, "Service representation, discovery and composition for E-marketplaces," in CoopIS Conf., September 2001, pp. 270–284.
22. IBM, BEA Systems, Microsoft, SAP AG, Sonic Software, and VeriSign, "WS-policy specification," http://www.ibm.com/developerworks/library/specification/ws-polfram, March 2006.
23. IBM, BEA Systems, Microsoft, SAP AG, Sonic Software, and VeriSign, "WS-policyattachment specification," http://www.ibm.com/developerworks/library/specification/ws-polatt, March 2006.
24. L. Kagal, M. Paolucci, N. Srinivasan, G. Denker, T.W. Finin, and K.P. Sycara, "Authorization and privacy for semantic Web services," IEEE Intelligent Systems, vol. 19, no. 4, pp. 50–56, 2004.
25. R. Khalaf, N. Mukhi, and S. Weerawarana, "Service-oriented composition in BPEL4WS," in WWW (Alternate Paper Tracks), May 2003.
26. C. Lee and S. Helal, "Context attributes: an approach to enable context- awareness for service discovery," in 2003 Symposium on Applications and the Internet (SAINT'03), 2003.

27. F. Leymann, D. Roller, and M.-T. Schmidt, “Web services and business process management,” IBM Systems Journal, vol. 41, no. 2, pp. 198–211, 2002.
28. L. Li and I. Horrocks, “A software framework for matchmaking based on semantic Web technology,” in WWW 2003 Conf., May 2003, pp. 331–339.
29. Z. Maamar, D Benslimane, and N. C. Narendra, “What can context do for Web services,” Communications of the ACM, to appear in 2006.
30. Z. Maamar, S.K. Mostefaoui, and H. Yahyaoui, “Toward an agent-based and context-oriented approach for Web services composition,” IEEE Transactions on Knowledge and Data Engineering (TKDE), vol. 17, no. 5, pp. 686–697, 2005.
31. Z. Maamar, S. Kouadri Mostefaoui, and Q.H. Mahmoud, “On personalizing Web services using context,” International Journal of E-Business Research, vol. 1, no. 3, 2005.
32. Z. Maamar, G. Kouadri Mostéfaoui, D. Benslimane, S. Sattanathan, and C. Ghedira, “Developing interoperable business processes using Web services and policies,” in 2nd International Conference on Interoperability for Enterprise Software and Applications, 2006.
33. D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara, “Bringing semantics to Web services: the OWL-S approach,” in First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004), California, July 2004.
34. S.A. McIlraith, T.C. Son, and H. Zeng, “Semantic Web services,” IEEE Intelligent Systems, vol. 16, no. 2, pp. 46–53, 2001.
35. M. Mecella, B. Pernici, and P. Craca, “Compatibility of e-services in a cooperative multi-platform environment,” in 2nd VLDB TES Workshop, September 2001, pp. 44–57.
36. B. Medjahed, B. Benatallah, A. Bouguettaya, A.H.H. Ngu, and A.K. Elmagarmid, “Business-to-Business Interactions: Issues and Enabling Technologies,” The VLDB Journal, vol. 12, no. 1, pp. 59–85, May 2003.
37. B. Medjahed and A. Bouguettaya, “A multilevel composability model for semantic Web services,” IEEE Transactions on Knowledge and Data Engineering (TKDE), vol. 17, no. 7, pp. 954–968, July 2005.
38. B. Medjahed and A. Bouguettaya, “Customized delivery of E-government Web services,” IEEE Intelligent Systems, vol. 20, no. 6, pp. 77–84, December 2005.
39. B. Medjahed, A. Bouguettaya, and A. Elmagarmid, “Composing Web services on the semantic Web,” The VLDB Journal, vol. 12, no. 4, pp. 333–351, November 2003.
40. G. Kouadri Mostefaoui, “Towards a conceptual and software framework for integrating context-based security in pervasive environments,” Ph.D. Dissertation, University of Fribourg, October 2004.
41. S. Narayanan and S.A. McIlraith, “Simulation verification and automated composition of Web services,” in WWW 2002 Conf., 2002, pp. 77–88.
42. M. Paolucci, T. Kawamura, T.R. Payne, and K. Sycara, “Semantic matching of Web services capabilities,” in First International Semantic Web Conference, 2002, pp. 333–347.
43. J. Pascoe, “The stick-e note architecture: extending the interface beyond the user,” in Intelligent User Interfaces, 1997, pp. 261–264.
44. T.R. Payne, M. Paolucci, , and K. Sycara, “Advertising and matching DAML-S service descriptions (position paper),” In Int’l Semantic Web Working Symp., 2001, pp. 76–78.
45. S. Ran, “A model for Web services discovery with QoS,” SIGecom Exchanges, vol. 4, no. 1, pp. 1–10, 2003.
46. N. Sriharee, T. Senivongse, K. Verma, and A.P. Sheth, “On using WS-policy, ontology, and rule reasoning to discover Web services,” in INTELLCOMM 2004, 2004, pp. 246–255.
47. K. Sycara, M. Klush, , and S. Widoff, “Dynamic service matchmaking among agents in open information environments,” ACM SIGMOD Record, vol. 28, no. 1, pp. 47–53, 1999.
48. A. Tsalgatidou and T. Pilioura, “An overview of standards and related technology in Web services,” Distributed and Parallel Databases, vol. 12, no. 3, pp. 135–162, November 2002.
49. R.M. Turner, “Context-mediated behavior for intelligent agents,” Int. J. Hum.-Comput. Stud., vol. 48, no. 3, pp. 307–330, 1998.
50. UDDI, “The universal description, discovery and integration (3.0),” http://www.uddi.org, February 2005.
51. K. Verma, R. Akkiraju, and R. Goodwin, “Semantic matching of Web service policies,” in Second International Workshop on Semantic and Dynamic Web Processes, 2005, pp. 79–90.
52. M. Wooldridge and N.R. Jennings, “Intelligent agents: theory and practice,” Knowledge Engineering Review, vol. 10, no. 2, pp. 115–152, 1995.

53. C. Wroe, R. Stevens, C.A. Goble, A. Roberts, and R.M. Greenwood, “A suite of daml+oil ontologies to describe bioinformatics Web services and data,” International Journal on Cooperative Information Systems, vol. 12, no. 2, pp. 197–224, 2003.
54. L. Zeng, B. Benatallah, A.H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, “QoS-aware middleware for Web services composition,” IEEE Trans. Software Eng., vol. 30, no. 5, pp. 311–327, 2004.