Distributed and Parallel Databases, 17, 179–206, 2005
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

A Dynamic Foundational Architecture for Semantic Web Services

BRAHIM MEDJAHED brahim@umich.edu
Department of Computer and Information Science, University of Michigan, Dearborn

ATHMAN BOUGUETTAYA athman@vt.edu
Department of Computer Science, Virginia Tech

Recommended by: Ahmed Elmagarmid

Abstract. The combination of *Web services* and *ontologies* is gaining momentum as the potential *silver bullets* for tomorrow's Web, i.e., the Semantic Web. We propose an architectural foundation for managing semantic Web services in dynamic environments. We introduce the concept of *community* to cater for an ontological organization of Web services. We develop an ontology called *community ontology* that serves as a "template" for describing communities of Web services. We also propose a peer-to-peer approach for managing communities in dynamic environments.

Keywords: semantic Web, Web services, ontologies, agents

1. Introduction

The world has seen a radical shift in economic, social, and political paradigms because of the introduction of the Web. The Web has enabled and created a vast array of opportunities that have transformed and transcended all types of activities. Before the advent of the Web, there was a general "hunger" for data, i.e., data was not readily or cheaply available. What we have witnessed since the Web has emerged, is a complete reversal of that trend where the average user is now literally submerged with too "much" data that is hard to make sense of. While search engines have tried to alleviate this problem, users are still locked into a paradigm that forces them to manually sift through already filtered data. The underlying issue has essentially been finding a strategy to turn Web data into *usable information*. This is where the *Semantic Web* comes into play. The Semantic Web aims at augmenting the Web with the necessary tools to allow data to be automatically *understood* by programs [2]. The building blocks of the envisioned Semantic Web, heavily focused on the use of standards, have already been laid down.

A key issue that has long been overlooked on the Web is the little or lack of broad leveraging of the tremendous investments in Web applications developed over the years to access Web data. These applications are typically customized to access corporate databases. An effective solution to this issue consists of eliciting programmatic access to these applications and viewing them as *first class* objects so they can be *reused* (*outsourced*) and *combined* (*composed*). *Web services* are a technology enabler proposed to materialize this vision [1, 10]. *Web services* are a recent addition to the Semantic Web to deal with the glut of Web applications. More formally, a *Web service* is a set of related functionalities that can be programmatically accessed through the Web. Examples of Web services span several application domains that include *e-government* (e.g., e-tax preparation) and *B2B e-commerce* (e.g., stock trading) [1, 13]. The maturity of XML-based Web service technologies such as SOAP, UDDI, and WSDL is one of the key factors contributing to the expected wide adoption of Web services [1].

Another concept that is taking the spotlight in the envisioned Semantic Web is the concept of *ontology*. An *ontology* is defined as a *formal* and *explicit* specification of a *shared conceptualization* [2, 7, 14]. Ontologies are increasingly seen as key to enabling semantics-driven data access and processing. They are expected to play a central role to empower Web services with semantics. The combination of these powerful concepts (i.e., Web services and ontologies) has resulted in the emergence of a new generation of Web services called *Semantic Web services*. Semantic Web services are poised to be the building blocks of tomorrow's Web, i.e., the Semantic Web [10]. Applications "exposed" as Web services would be *understood*, *shared*, and *invoked* by *automated* tools.

Semantic Web services have spurred an intense activity in industry and academia to address challenging research issues such as the *description*, *selection*, *monitoring*, and *composition* of Web services. The diversity of these issues calls for the design and development of a *Web Service Management System* where Web services would be treated as first-class objects that can be manipulated as if they were pieces of data. The research presented in this paper is part of a comprehensive framework we are developing for enabling the *automatic composition* of Semantic Web services. Web service composition refers to the process of combining several Web services to provide a *value-added* service, called *composite* service [1, 10]. Paramount in this framework is the semantics of Web services. The automatic composition of Web services raises several challenging issues including the *meaningful* organization and description of Web services, *selection* of relevant services, *composability* of participant services (i.e., automatically checking whether they can actually work together), and *generating* composite service descriptions. Our focus in this paper is on the issue of *organizing*, *describing*, and *managing* semantic Web services.

The semantic organization and description of Web services is an important requirement for deploying the envisioned Semantic Web. The number of Web services available on the Web is increasing fast. Service providers maybe located in different places across the globe. The large scale, dynamics, and heterogeneity of Web services may hinder any attempt for "understanding" their semantics and hence managing them. To address this issue, we propose a Semantic Web centered framework for organizing and describing Web services. We introduce the concept of *community* to cater for an ontological organization of Web services. Services are clustered into communities based on their domain of interest. We develop an ontology called *community ontology* that serves as a "template" for describing communities of Web services. A community provides a set of *generic* operations that can be used "as is" or customized by underlying services. Service providers identify the community of interest and register their service with it. The registration process is handled by a network of *software agents* associated to service and community providers. An agent is a piece of software capable of acting proactively to accomplish tasks on behalf of its consumers (e.g., service providers) [11].

The rest of this paper is organized as follows. In Section 2, we introduce the concept of *community* to cater for an ontological organization of the service space. In Section 3, we describe operational features of communities via *generic operations*. In Section 4, we present the technique used for registering Web services with a community. In Section 5, we propose a peer-to-peer approach for managing communities of Web services. In Section 6, we describe our implementation. In Section 7, we give a brief survey of the related work. We finally provide concluding remarks in Section 8.

2. An ontological framework for semantic Web services

While the outcomes of our approach are generic enough to be applicable to a wide range of applications, we specifically target the area of government social services providing benefits for senior citizens as a case study. Typically, a large set of services is offered to help senior citizens including *FasTran* (transportation), *Meals on Wheels* (meal deliverers), *Meals Providers* (restaurants), *Senior Activity Center* (social club for senior citizens). The number of e-government Web services may be large. Some of those services are offered by specialized agencies such as the *Department for the Aging*. Others are subcontracted from outside organizations including state and federal government agencies (e.g., Department of Health and Human Services), businesses (e.g., restaurants participating in a subsidized government program), volunteer centers (e.g., used by Meals on Wheels service), and non-profit organizations (e.g., American Red Cross).

2.1. Ontological support for communities

Combining the emerging Semantic Web concepts of Web services and ontology are at the core of our approach. To cater for an ontological organization and description of Web services, we propose the concept of *community*. A *community* is a "container" that clusters Web services based on a specific area of interest (e.g., disability, adoption). All Web services that belong to a given community share the same area of interest. Communities provide descriptions of desired services (e.g., providing interfaces for insurance services) without referring to any actual service.

We define a *metadata* ontology, called *community ontology*, for creating communities of Web services. *Metadata* ontologies provide concepts that allow the description of other concepts (communities and Web services in our case) [7]. Figure 1 summarizes the proposed model for semantic Web services. Communities are instances of the community ontology. They are created by *community providers* which use the *community ontology* as a template. Community providers are generally groups of government agencies, non-profit organizations, and businesses that share a common domain of interest. For example, the Department for the Aging and other related agencies, such as the Department of Health, would define a community that provides healthcare benefits for senior citizens. A community is itself a service that is created, advertised, discovered, and invoked in the same way "regular" Web services are. The providers of a community assign values to the attributes

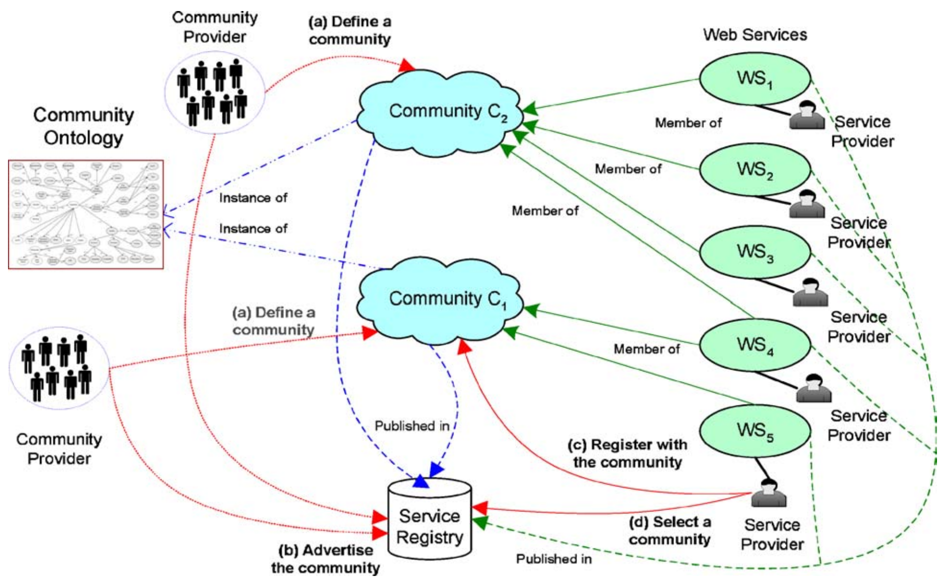

Figure 1. The proposed model for Semantic Web services.

and concepts of the community ontology (figure 1: step a). Communities are published in a registry (e.g., UDDI) so that they can be discovered by service providers (figure 1: step b). *Service providers* (e.g., `medicare` provider) identify the community of interest (figure 1: step c) and register their services with it (figure 1: step d). A Web service may belong to different communities. For example a composite service (WS$_4$ in figure 1) may outsource operations that have different domains of interest (e.g., "healthcare" and "elderly"). Since these operations belong to two different communities, the composite service is registered with the "healthcare" and "elderly" communities (C$_1$ and C$_2$ in figure 1). End-users (e.g., case officers) select a community of interest and invoke its operations. Each invocation of a community operation is translated into the invocation of a community member's operation.

We use the emerging DAML+OIL language for describing the proposed ontology [9]. DAML+OIL builds on earlier Web ontology standards such as RDF and RDF Schema and extends those languages with richer modeling primitives (e.g., cardinality) [9]. Our approach is not dependent on DAML+OIL; other Web ontology standards (e.g., OWL) could be used to describe the community ontology.

2.2. *Structure of a community*

A community C_i is formally defined by a tuple (*Identifier$_i$*, *Category$_i$*, *G-operation$_i$*, *Members$_i$*). The *identifier$_i$* clause contains a unique *name* and a text *description* that summarizes C_i's features (figure 2). *Category$_i$* describes the area of interest of the community.

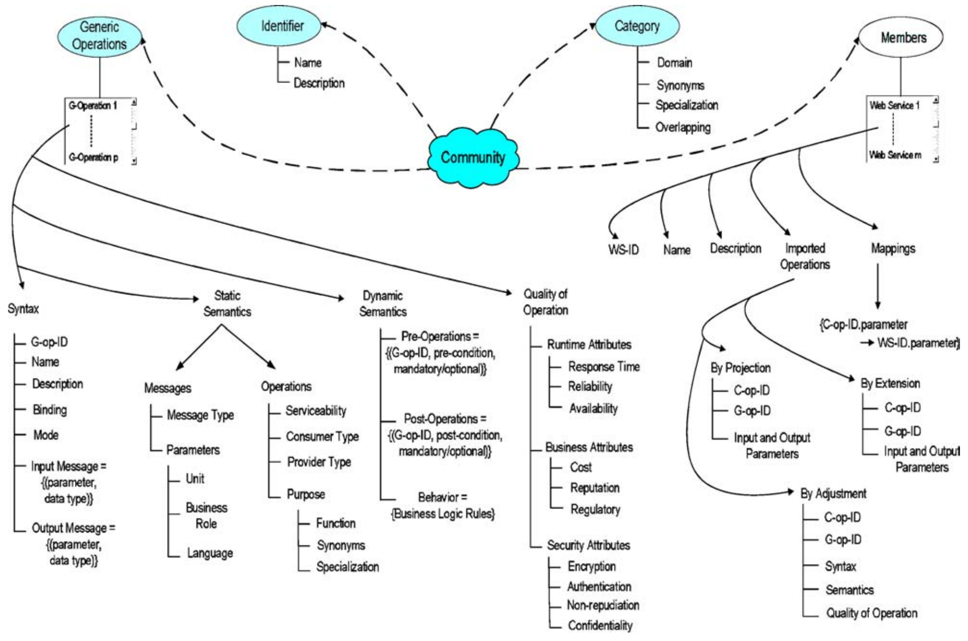

Figure 2. The general structure of a community.

All Web services that belong to C_i have the same category as C_i's. C_i is accessible via a set of operations called *generic operations*. Those are specified in the *G-operation$_i$* clause. Generic operations are "abstract" operations that summarize the major functions needed by C_i's members. Community providers define generic operations based on their expertise on the corresponding area of interest that is, C_i's category. The term "abstract" means that no implementation is provided for generic operations. Community providers only define an interface for each generic operation op_{ik}. This interface could subsequently be used and implemented by community members (i.e., actual Web services) interested in offering op_{ik}. We say that those members *support* or *import* op_{ik}. The execution of op_{ik} hence refers to the execution of an actual operation offered by a member that support op_{ik}. The *Members$_i$* clause refers to the list of C_i's members. By being members of C_i, Web service providers "promise" that they will be*supporting* one or several of C_i's generic operations. In this section, we focus on describing the category clause and community ontology. Details about generic operations and community members are given in Sections 3 and 4 respectively.

The *category* of a community C_i is formally defined by a tuple (*Domain$_i$*, *Synonyms$_i$*, *Specialization$_i$*, *Overlapping$_i$*). *Domain$_i$* gives the area of interest of the community (e.g., "healthcare"). It takes its value from a taxonomy for domain names. For flexibility purposes, different communities may adopt different taxonomies to specify their category. We use XML namespaces to prefix categories with the taxonomy in which they are defined. Simply put, XML namespaces provide a method for qualifying element and attribute names used in XML documents by associating them with URI references. *Synonyms$_i$* contains a set of alternative domain names for C_i. For example “medical” is a synonym of “healthcare”. Values assigned to this attribute are taken from the same taxonomy as the one used for domains. *Specialization*$_i$ is a set of characteristics of the C_i’s domain.For example, “insurance” and “children” are specialization of “healthcare”. This means that C_i provides health insurance services for children. Communities are generally not independent. They are linked to each other via *inter-ontology* relationships. These relationships are specified in the *Overlapping*$_i$ attribute. *Overlapping*$_i$ contains the list of categories that *overlap* with C_i’s category. It is used to provide a peer-to-peer topology for connecting communities with “overlapping” categories. We say that *category*$_i$ and *category*$_j$ *overlap* if they are related to each other. For example, an operation that belongs to a community whose domain is *family* may be combined with another operation that belong to a community whose domain is *insurance*. This would enable providing health insurance for needy families. It should be noted that it is the responsibility of the community providers to identify related categories and assign them to the overlapping attribute.

2.3. Generic operations

A generic operation is defined by a set of *functional* and *non-functional* attributes. *Functional* attributes describe *syntactic* and *semantic* features of generic operations. *Syntactic* attributes represent the structure of a generic operation. An example of syntactic attribute is the list of input and output parameters that define the operation’s messages. *Semantic* attributes refer to the meaning of the operation or its messages. We consider two types of semantic attributes: *static* and *dynamic* semantic attributes. *Static* semantic attributes (or simply static attributes) describe non-computational features of generic operations. Those are semantic attributes that are generally independent of the execution of the operation. An example of static attribute is the operation’s category. *Dynamic* semantic attributes (or simply dynamic attributes) describe computational features of generic operations. They generally refer to the way and constraints under which the operation is executed. An example of dynamic attribute is the business logic of the operation i.e., the results returned by the operation given certain parameters and conditions.

Non-functional attributes, also called *qualitative* attributes, include a set of metrics that measure the quality of the operation. Examples of such attributes include time, availability, and cost. Two services providers that support the same generic operation may have different values for their qualitative attributes. Non-functional attributes model in fact the competitive advantage that Web services may have on each other.

While defining a community, community providers assign values to part of the attributes of their generic operations. The rest of the attributes are assigned either by service providers or third parties during the registration of Web services with C_i. For example the types of input and output messages (e.g., purchase order, registration confirmation) are define by community providers. The cost (dollar amount) of executing an operation is service-specific and hence defined by the service provider. The other qualitative attributes (e.g., response time, availability) are assigned by third parties (e.g., trusted parties). The way those parties determine the values to be assigned (e.g., via monitoring) is out of the scope of this paper. It is worth noting that the values of some attributes may be assigned by both community and service providers. For example, the content of input/output messages is given by community providers. However, service providers may modify this content by adding and/or removing parameters to input and output messages.

2.4. Community members

Service providers can, at any time, select a community of interest (based on categories) and register their services with it. The registration process requires giving an *identifier* (*WS-ID*), *name*, and *description* for the Web service. The *identifier* takes the form of a unique UUID. The *description* summarizes the main features of the Web service. Service providers specify the list of generic operations supported by their service through the *imported* attribute. We define three techniques for importing generic operations: *projection*, *extension*, and *adjustment*. Details about these techniques are given in Section 4. The invocation of an importedoperation is translated into the invocation of an "actual" service operation. The correspondence between imported and "actual" operations is done through the *mapping* attribute. For each imported operation, the provider gives the ID of the corresponding "actual" operation. It also defines a one-to-one mapping between the imported operation's parameters and "actual" operation's parameters. Defining mappings between parameters enables the support of "legacy" Web services. Providers do not need to change the message parameters in their actual service codes.

Assume that a service provider *SP* offers a given operation *op*. The following three cases are then possible: (i) If there is a community C_i that contains a generic operation op_{ik} similar to *op*, *SP* would import op_{ik} "as is"; (ii) If there is a community C_i that contains a generic operation op_{ik} "closely" similar to *op* (e.g., *op* has less input parameters than defined in op_{ik}), *SP* would import op_{ik} using *projection*, *extension*, and/or *adjustment* technique; (iii) If no community has an operation similar or "closely" similar to *op*, *SP* would define a new community C_j that has *op* as a generic operation and *SP*'s service as a member. The latter case is similar to the "traditional" WSDL/UDDI/SOAP model where service providers create descriptions for their services. The difference is that, in our case, *SP* instantiates the attributes and concepts of the community ontology while in the "traditional" model, providers define their service descriptions from scratch.

3. Operational description of communities via generic operations

As mentioned previously, a generic operation is described at four different levels: *syntactic*, *static semantic*, *dynamic semantic* and *qualitative* levels. In this section, we give a detailed description of generic operation attributes for each of those levels.

3.1. Syntactic attributes

We identify two levels for syntactically describing a generic operation: *operation* and *message* levels. The operation level contains attributes that describe the operation (e.g., *name*, *description* of the operation). The message levels caters for the syntactic description of messages exchanged by the operation (e.g., *name*, and *data type* of a message parameter).

3.1.1. Operation syntax. A generic operation has a unique identifier called *G-op-ID*. It takes the form of a *Universally Unique ID* (UUID). A UUID is an identifier that is unique across both space and time [1]. The operation also has a *name* and a text *description* thatsummarizes the operation's features. The *binding* defines the message formats and protocols used to interact with the operation. An operation may be accessible using several bindings such as *SOAP/HTTP*, and *SOAP/MIME*. The binding of an operation is assigned by the service provider. This is in contrast to the rest of syntactic attributes whose values are pre-defined by community providers. Indeed, the binding attribute is dependent on the way the generic operation is implemented at the service provider side. A provider may offer SOAP/HTTP access to a generic operation supported by its Web service while another provider may prefer to use SOAP/MIME for the same operation.

The *mode* of an operation refers to the order according to which its input and output messages are sent and received. It states whether the operation initiates interactions or simply replies to invocations from other services. We define two operation modes: *In/Out* or *Out/In*. One of this values is assigned by community providers to each operation. *In/Out* operation first receives an input message by a client, process it (locally or forward it to another service), and then returns an output message to the client. *Out/In* first sends an output message to a server and receives an input message as a result. An example of *In/Out* operation is `checkEligibilityWIC` which checks citizen's eligibility for a nutrition program for pregnant women (WIC or Women Infant and Children). As specified in WSDL standard, some operations may be limited to an input or an output message [1]. For example, `expirationWIC` is an operation that automatically notifies citizens about the termination of their eligibility period for WIC programs. This operation obviously does not require any input message. However, such operations may be considered as *In/Out* or *Out/In* operations where the input or output message is empty. Hence, without loss of generality, we focus in the proposed framework on the aforementioned operation modes.

3.1.2. Message syntax. Each input or output message contains one or more parameters. A parameter has a *name* and *data type*. We use the set *In-Out* (*G-op-ID*) to refer to the list of input and output parameters' names. The data type gives the range of values that may be assigned to the parameter. We use XML Schema's *built-in* data types as the typing system. *Built-in* types are pre-defined in the XML schema specification. They can be either *primitive* or *derived*. Unlike *primitive* types (e.g., `string`, `decimal`), *derived* types are defined in terms of other types. For example, `integer` is derived from the `decimal` primitive type. Although message parameters are predefined by community providers, service providers have the ability to add new parameters, remove some, or change the content (e.g., data type) of pre-defined parameters. In Section 4, we give more details about the way service providers modify the content of operation messages.

3.2. Static semantic attributes

The static semantics of a generic operation refers to a set of attributes whose content is independent of the execution of the operation. It should not only specify the semantics of the operation itself (e.g., what does the operation do) but also the semantics of all messages handled by the operation. In the rest of this section, we define static semantics at two levels: the message parameter and operation levels.

3.2.1. Operation semantics. The static semantic of an operation is defined by the following attributes: *serviceability*, *provider type*, *consumer type*, *purpose*, and *category*. *Serviceability* gives the type of assistance provided by the operation. Examples of values for this attribute are "cash", "in-kind", "informational", and "educational". TANF (Temporary Assistance for Needy Families) is an example of welfare program that provides financial support to needy families. A food stamp is an example of in-kind support available to indigent citizens. Returning the list of senior activity centers is an example of informational support. Enhancing communication skills of visuallyimpaired people is an example of educational support. Other types of support may bementioned by assigning the value "other" to this attribute.

A generic operation may be supported viaone or several *provider types*. A provider may be governmental ("federal", "state", "local", and "tribal") or non-governmental ("non-profit" and "business") agencies. For example, `nursingHome` may be provided by the Department of Aging (government agency) and Red Cross (non-profit). The *consumer type* property specifies the group of citizens (e.g., children, pregnant women) that are eligible to the operation's welfare program. Different groups may be eligible for the same benefit. For example, WIC (Women, Infant, and Children) is a program for pregnant women, lactating mothers, and children.

Each generic operation performs a certain functionality for a specific area of interest. This is specified through the *purpose* and *category* attributes respectively. An operation inherits the category of the community in which it is defined. Hence, all operations that belong to the same community share the same category. The *purpose* attribute describes the goal of the operation. It is defined by three attributes: *function*, *synonyms*, and *specialization*. The *function* describes the businessfunctionality offered by the operation. Examples of functionsare "eligibility" and "registration". *Synonyms* and *specialization* attributes work as they do for categories.

3.2.2. Message semantics. Messages must be semantically described so that they can be "correctly" interpreted by service providers and consumers. We associate a *message type* $\mathcal{MT}$ to each message. $\mathcal{MT}$ gives a general description of the content represented by the message. For example, a message may represent a purchase order or an invoice. *Vertical* ontologies are the ideal concept to describe the type of message. *Vertical* ontologies capture the knowledge valid for a particular domain such as medical, mechanic, chemistry, and electronic [7]. An example of such ontology is *RosettaNet's PIPs* (*Partner Interface Processes*) [3].

The message type does not capture the whole semantics of a message. Indeed, it is important to describe the semantics of each parameter belonging to that message. Message parameters are so far represented through their name and data type. While data types are an important feature of message parameters, they do not capture their semantics. For example, an `eligibility period` parameter may be specified in days, weeks, or months. We identify three attributes to model the semantics of message parameters: *business role*, *unit*, and *language*. These attributes are pre-defined by community providers. The *business role* gives the type of information conveyed by the message parameter. For example, an `address` parameter may refer to the first (street address and unit number) or second (city and zip code) line of an address. Another example is that of a `price` parameter. It may represent a total price or price without taxes. Business roles take their values from a pre-defined taxonomy. Every parameter would have a well-defined meaning according to that taxonomy. An example of such taxonomy is *RosettaNet*'s business dictionary [3]. It contains a common vocabulary that can be used to describe business properties. For example, if the `price` parameter has an "extendedPrice" role (defined in *RosettaNet*), then it represents a "total price for a product quantity". For flexibility purposes, different community providers may adopt different taxonomies to specify their parameters' business roles. As for categories, we use XML namespaces to prefix business roles with the taxonomy according to which they are defined.

The *unit* attribute refers to the measurement units in which the parameter's content is provided. For example, a `weight` parameter may be expressed in "Kilograms" or "Pounds". A `price` parameter may be in "US Dollars" or "Euro". We use standard measurement units (length, area, weight, money code, etc.) to assign values to parameters' units. If a parameter does not have a unit (e.g., `address`), its unit is equal to "none". The content of a message parameter may be specified in different languages. For example, a `profession` parameter may be expressed in English or Spanish. A `translation` operation make take as input, an English word (input parameter) and returns as output, its translation in Urdu (output parameter). We adopt the standard taxonomy for languages to specify the value of this attribute. In contrast to the business role whose content is handled only by community providers, the unit and language attributes may changed by service providers. Service providers should have the flexibility to support more units and languages than those specified by community providers. For example, a service provider may decide to support a *weight* parameter in both "Kilograms" and "Pounds" although the community providers specified "Pounds" as the only measurement unit for this parameter. We give below a formal definition of a message parameter. The definition includes both syntactic and semantic attributes. Each message (input or output) is defined as a set of such parameters.

Definition 1 (Message parameter). A message parameter P is defined as a tuple $(\mathcal{T}, \mathcal{R}, \mathcal{U}, \mathcal{L})$ where $\mathcal{T}$ is the parameter's data type (in XML Schema), $\mathcal{R}$ is its business role taken from a taxonomy for business roles, $\mathcal{U}$ gives P's unit of measurement, and $\mathcal{L}$ is the set of languages according to which P may be expressed.

3.3. Dynamic semantics

Dynamic semantics allows the description of attributes related to the execution of generic operations. Those attributes may relate the execution of an operation op_{ik} to the execution of other operations (*inter-operation* attributes) or describe features inherent to the execution of op_{ik} (*intra-operation* attributes). The execution of an operation op_{ik} generally goes through four major observable states: *Ready*, *Start*, *Active*, and *end*. The execution of op_{ik} is in the *Ready* state if the request for executing op_{ik} has not been made yet. The *Start* state means that op_{ik} execution has been initiated. It refers to one of the following events: (i) an input message is sent to op_{ik} if op_{ik}'s mode is *In/Out*; or (ii) an output message has been sent from op_{ik} if op_{ik}'s mode is *Out/In*. We say that op_{ik} is in the *Active* state if op_{ik} has already been initiated and the corresponding request is being processed. After processing the request, the operation reaches the *End* state during which results are returned. It refers to one of the following events: (i) an output message is sent to the client if op_{ik}'s mode is *In/Out*; or (ii) an input message is received from the server if op_{ik}'s mode is *Out/In*. We define a precedence relationship between states, noted $\rightarrow_t$, as follows: $S_1 \rightarrow_t S_2$ if S_1 occurs before S_2.

3.3.1. Pre-operations. Executing a Web service operation may require going through a pre-defined process that involvesthe execution of several operations called *pre-operations*.This pre-defined process may be dictated by government regulations. For example, senior citizens must first register with a an Area Agency on Aging via `checkRegistration` operation before applying for any welfare program. They may also reflect the business logic of the Web service. For example, senior citizens must order a meal from a participating restaurant via the `orderMeal` operation before requesting its delivery through the `mealsOnWheels` operation.

Let us consider two generic operations op_{ik} and op_{jl} that belong to the same or different communities. We say that op_{ik} is a *pre-operation* of op_{jl} if the invocation of op_{jl} is preceded by the execution of op_{ik}. We call op_{ik} and op_{jl} *source* and *target* operations respectively. An operation may have several pre-operations. It may also be the source (i.e., pre-operation) of several operations. We give below a formal definition of the *pre-operation* relationship.

Definition 2 (Pre-operation). Let op_{ik} and op_{jl} be two generic operations. op_{ik} is a *pre-operation* of op_{jl} if $\textit{End}(\text{op}_{ik}) \rightarrow_t \textit{Ready}(\text{op}_{jl})$.

The definition of a pre-operation relationship includes a source operation op_{ik}, target operation op_{jl}, and the *condition* and *mandatory* attributes. The *condition* is a predicate over op_{ik}'s input and output parameters. op_{jl} can be invoked only if all its pre-operations reached their End state and their conditions are true. If no condition is specified for a given pre-operation then the default value is “true”. For example, `mealsOnWheels` is executed only if the `orderMeal` operation has been approved. The *mandatory* attribute takes boolean values and specifies whether the executing the source operation is mandatory or optional. If this attribute is true then the relationship between op_{ik} and op_{jl} is obligatory. Otherwise, it is *recommended*. For example, senior citizens *must* have ordered meals from a restaurant before requesting home delivery from a volunteer center. However, it is recommended to get the list of participating (`participatingRestaurants`) restaurants before ordering a meal. Indeed, users can directly order a meal from a restaurant if they already know that restaurant's name.

3.3.2. Post-operations. The execution of a given operation may trigger the invocation of other operations called *post-operations*. For example, a pregnant women that registers successfully for a food check program women (`registerFoodCheck`) is required to register for a nutritional counseling course (`registerNutritionCourse`). We say that op_{ik} is a *post-operation* of op_{jl} if the termination of op_{jl} precedes the invocation of op_{ik}. We call op_{jl} and op_{ik} *source* and *target* operations respectively. An operation may have several post-operations. It may also be the target (i.e., post-operation) of several operations. Note that if op_{ik} is a pre-operation of op_{jl} then op_{jl} is not necessarily a post-operation of op_{ik}. For example, `checkRegistration` is pre-operation of `orderMeal`. However, `orderMeal` is not a post-operation of `checkRegistration`. Indeed, users do not need to order meals whenever their registration with the Department on the Aging is checked. We give below a formal definition of the *post-operation* relationship.

Definition 3 (Post-operation). Let op_{ik} and op_{jl} be two generic operations. op_{ik} is a *post-operation* of op_{jl} if $\textit{End}(\text{op}_{jl}) \rightarrow_t \textit{Ready}(\text{op}_{ik})$.

As for pre-conditions, we associate a *condition* and *mandatory* attribute to each post-operation relationship. A target operation enters the initiation states if *at least* one of its source operations has reached its End state and the corresponding condition is true. A post-operation may also bemandatory or optional. For example, a pregnantwomen that registers for a food check program women (`registerFoodCheck`) *must* also register for nutritional counseling course by invoking `registerNutritionCourse` (mandatory = true). The post-operation `register` is optional (mandatory = false). Indeed, citizens do not necessarily have to register with the Department on the Aging if they are not willing to do so.

3.3.3. Community diagram. Pre and post-operations provide means to specify pre-defined business processes within a community. In the case of e-government, those business processes are mostly driven by government regulations and laws. Several process models have been proposed in the literature such as *Petri-nets*, *statecharts*, and *π-calculus*. In our approach, we adopt *UML activity diagrams* as to model pre-operation and post-operation relationships within a community [8]. We refer to such diagram as a *community diagram*. Activity diagrams are the most widely used process modeling techniques both in conventional interaction technologies (e.g., workflows) and Web services [1]. The reason for their success is their ease-of-use and simplicity for modeling business processes. Several tools (e.g., *Rational Rose*) are available for designing business processes using activity diagrams. Additionally, the *Unified Modeling Language* (*UML*) has become the *de facto* standard for representing application architecture and design models. Finally, activity diagrams model orchestrations by specifying which actions should be performed, from the beginning of the execution to the end. This seems to be the most natural way in which community and service providers think of a process [1].

Activity diagrams show the flow of *activities* in a business process. In our approach, each activity represents a generic operation. Generic operation within a diagram may belong to different communities. In this case, the community name prefaces the G-op-IDs. We refer to such pre/post operations as *remote* pre/post operations. We depict in figure 3 part of the diagramfor the `elderly` community. The filled circle is the starting point of the

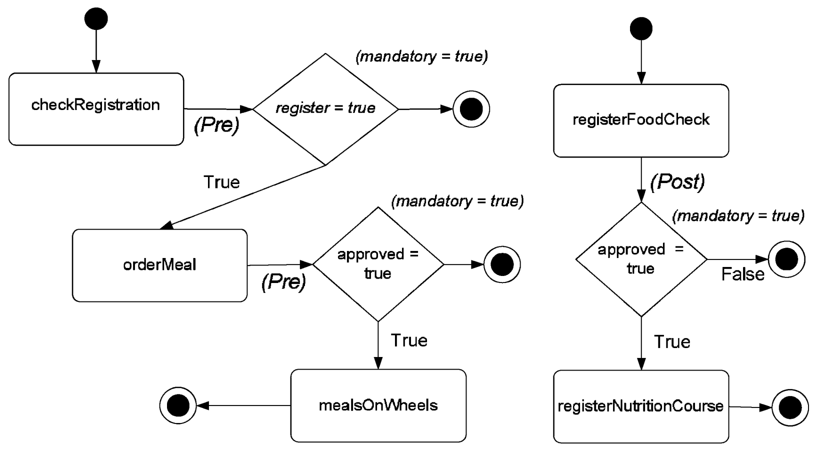

Figure 3. An example of community diagram.

diagram. The filled circle with a border is the ending point. Each edge in the diagram is labeled with a *relationship* attribute. This attribute takes one of the values "Pre" or "Post" to specify whether the edge models a pre or post-operation. For example, the edge `checkRegistration` $\rightarrow$ `orderMeal` models a pre-operation relationship. It specifies that users should execute `checkRegistration` before initiating the execution of `orderMeal`. Community diagrams may also indicate that one operation *conditionally* follows another. For example, the *diamond* between the `checkRegistration` and `orderMeal` operations states that the the value returned by the `register` parameter should be "true" before invoking `orderMeal`. The edge `registerFoodCheck` $\rightarrow$ `registerNutritionCourse` models a post-operation relationship. It mentions that all citizens that successfully apply for a food check program should register for a nutrition course. Conditional constructs (i.e., diamonds) are labeled with a *mandatory* attribute defined as in Section 3.3.1. Community diagrams may also model *parallelism* via *fork* and *join* constructs. A *fork* is represented by a black bar with one flow going into it and several leaving it. It denotes the beginning of parallel processing. A *join* is depicted by a black bar with several flows entering it and one leaving it. It denotes the end of parallel processing.

3.3.4. Behavior. The *behavior* of a generic operation op_{ik} refers to the outcome expected after executing op_{ik} given a specific condition. It is defined by a set of *business logic rules* where each rule R_{ik}^{m} has the following format:

$$R_{ik}^{m} = \frac{\left(\text{PreParameters}_{ik}^{m}, \text{PreCondition}_{ik}^{m}\right)}{\left(\text{PostParameters}_{ik}^{m}, \text{PostCondition}_{ik}^{m}\right)}$$

$\text{PreParameters}_{ik}^{m}$ and $\text{PostParameters}_{ik}^{m}$ are sets of parameters. Each parameter is defined by name, data type, business role, unit, and language as stated in Definition 1. The elements of PreParameters$^m_{ik}$ and PostParameters$^m_{ik}$ generally refer to op$_{ik}$'s input and output parameters. However, they may in some cases refer to parameters that are neither input nor output of op$_{ik}$. For example, assume that the *address* of every citizen registered with the Department on the Aging is stored in the department's database. In this case, this parameter should not be required as input for the `orderMeal` operation since its value could be retrieved from the database. PreCondition$^m_{ik}$ and PostCondition$^m_{ik}$ are predicates over the parameters in PreParameters$^m_{ik}$ and PostParameters$^m_{ik}$ respectively. The rule R^m_{ik} specifies that if PreCondition$^m_{ik}$ holds when the operation op$_{ik}$ starts, then PostCondition$^m_{ik}$ holds after op$_{ik}$ reaches its End state. If PreCondition$^m_{ik}$ does not hold, there are no guarantees about the behavior of the operation. Preconditions generally specify relationships between input values. Similarly, post-conditions generally specify relationships between the returned values. The following is an example of the pre and post condition of a rule associated with the operation `registerFoodCheck`:

PreCondition: $(income < 22{,}090) \wedge (familySize \geq 2) \wedge (zipCode = 22044)$
PostCondition: $(approved = true) \wedge (duration = 6)$

The rule uses the parameters *income* (unit = {year, US dollar}), *familySize*, and *zipcode* in the pre-condition. The attributes *approved* and *duration* (unit = {month}) are used in the post-condition. The rule specifies that citizens with a yearly income less than 22,090 US dollars and a minimum household size 2 are eligiblefor food checks and the eligibility period is 6 months.

3.4. Qualitative properties

Multiple Web services that belong to the same community may import the same generic operation. It is hence important to define a set attributes that help select the "best" Web service supporting a given functionality. For this purpose, we define a *Quality of Operation* (QoP) model based on a set of *qualitative* attributes that are transversal to all operations such as the cost and response time.

The international quality standard ISO 8402 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" [12, 15]. Wedefine QoP as a set of non-functional attributes that may impact the quality of the operations imported by a Web service. There are many QoP attributes important to Web services operations. We organize them into three groups of quantifiable attributes based on type of measurement performed by each attribute: *run-time*, *business*, and *security*. Note that other attributes and groups may be added to our QoP model without altering the generality of our approach.

Run-time attributes. These attributes enable the measurement of properties that are related to the execution of an operation op_{ik}. We identify three run-time attributes: *response time*, *reliability*, and *availability*. The *response time* measures the expected delay in seconds between the moment when op_{ik}) enters the Start state (i.e., op_{ik} is initiated) and reaches the End state (i.e., op_{ik} gets or sends the results). Time(op_{ik}) is computed using the expression $\text{Time}_{\text{process}}(op_{ik}) + \text{Time}_{\text{results}}(op_{ik})$. This means that the response time includes the time to process the operation ($\text{Time}_{\text{process}}$) and the time to transmit or receive the results ($\text{Time}_{\text{results}}$). The *reliability* of op_{ik} is the ability of the operation to be executed within the maximum expected time frame. Reliability(op_{ik}) is computed based on historical data about previous invocations of the operation using the expression $\text{N}_{\text{success}}(op_{ik})/\text{N}_{\text{invoked}}(op_{ik})$ where $\text{N}_{\text{success}}(op_{ik})$ is the number of times that the operation has been successfully executed within maximum expected time frame and $\text{N}_{\text{invoked}}(op_{ik})$ is the total number of invocations. The *availability* is the probability that the operation is accessible. Availability(op_{ik}) is measured by the expression UpTime(op_{ik})/TotalTime(op_{ik}) where UpTime is the time op_{ik} was accessible during the total measurement time TotalTime.

Business attributes. These attributes allow the assessment of an operation op_{ik} froma business perspective. We identify three business attributes: *cost*, *reputation*, and *regulatory*. The *cost* gives the dollar amount required to execute op_{ik}. The *reputation* of op_{ik} is a measure of the operation's trustworthiness. It mainly dependson users' experiences on invoking op_{ik}. Users are givena range to rank Web service operations (e.g., between 1 and 10). The lowest value refer to the best ranking. Different users may have different opinions on the same operation. The reputation of op_{ik} is defined by the average ranking given by users to the operation. Reputation(op_{ik}) is computed by the expression $\sum_{u=1}^{n} Ranking_u(op_{ik})/n$, where $Ranking_u$ is the ranking by user u and n is the number of the times the operation has been ranked. The *regulatory* property is a measure of how well op_{ik} is aligned with government regulations. Regulatory(op_{ik}) is value within a range (e.g., between 1 and 10). The lowest value refer to an operation that is highly compliant with government regulations.

Security attributes. These attributes describe whether the operation op_{ik} is compliant with security requirements. Indeed, service providers collect, store, process, and share information about millions of users who have different preferences regarding security of their information. We identify four properties related to security and privacy: *encryption*, *authentication*, *non-repudiation*, and *confidentiality*. *Encryption* is a boolean that indicates whether op_{ik}'s message aresecurely exchanged (using encryption techniques) between servers and clients. *Authentication*is a boolean that states whether op_{ik}'s consumers (users and other services) are authenticated (e.g., through passwords). *Non-repudiation* is a boolean that specifies whether participants (consumers and providers) can deny requesting or delivering the service after the fact. The *confidentiality* attribute indicates which parties are authorized to access the operation's input and output parameters. Confidentiality (op_{ik}) contains op_{ik}'s input and output parameters that should not be divulged to external entities (i.e., other than the service provider). If a parameterdoes not belong to Confidentiality(op_{ik}), then no confidentiality constraint is specified on that parameter. Assume that *confidentiality (*$\text{op}_{ik} = \{$`SSN`, `salary`$\}$ where `SSN`, `salary` are two op_{ik}'s input parameters. The content of this attribute states that those two parameters are kept private by op_{ik}'s provider.

4. Registering Web services with communities

Registering A Web service with a community refers to the process of importing generic operations. The invocation of an imported operation is translated into the invocation of an "actual" service operation. The correspondence between imported and "actual" operations is done through the *mapping* attribute. For each imported operation, the provider gives the ID of the corresponding "actual" operation. It also defines a one-to-one mapping between the imported operation's parameters and "actual" operation's parameters. Defining mappings between parameters enables the support of "legacy" Web services. Providers do not need to modify the message parameters in their actual service codes.

4.1. The Web service registration process

The registration process is handled by a network of software agents associated to service and community providers. *Member* and *community agents* (MA_j and CA_i) are attached to each service and community provider (SP_j and CP_i) respectively (figure 4). MA_j handles the registration of SP_j's Web services with the community C_i. SP_j registers its service *WS-ID* using the following *registration statement*:

Register Service *WS-ID* **With Community** C_i
  Name *service-name*
  Description *service-description*
  [**Imported Generic** *G-op-ID*
    <importing statements>
    Mappings With Actual *Op-ID*
      [*G-op-ID.<parameter>* **Maps To** *Op–ID.<parameter>*]$^+$]$^+$

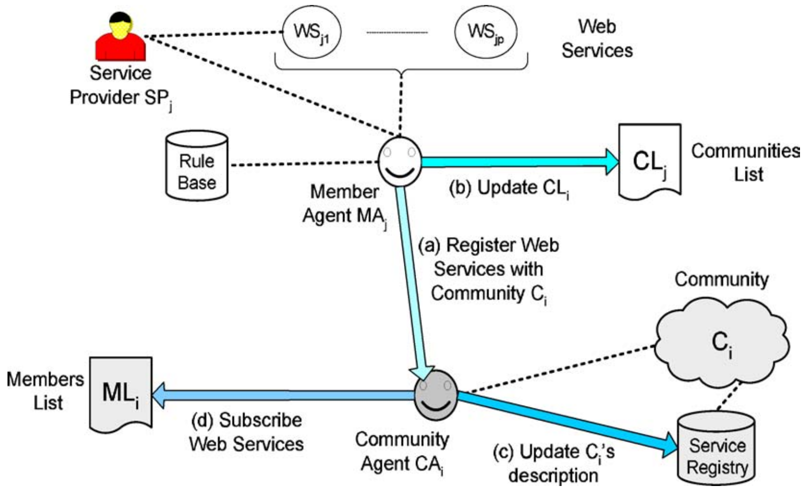

Figure 4. The web service registration process.

The clauses in the aforementioned statement correspond to the different attributes defined for service members within the community ontology. The <importing statements> is a sequence of statements for importing generic operation. A Web service may import several generic operations as stated by the "+" iteration symbol.

MA_j parses the registration statement and sends a registration message *SP_Register*(*WS-ID,name,desc,imported*) to CA_i (figure 4, step (a)). The message includes the service ID (*WS-ID*), name, description, and the list of imported operations. The *imported* set is equal to $\{$(*G-op-ID,mappings,AttSet,NotImpSet*)$\}$. It includes the mappings of G-op-ID (ID of the imported operation) with actual service operation. *AttSet* is the set of G-op-ID's attributes with their values as assigned by SP_j. It is defined by the set $\{$(*attribute,value*)$\}$. *NotImpSet* is the list of non-imported attributes. MA_j maintains a *Communities List* CL_j and *Rule Base* RB_j. CL_j is the list of communities with which SP_j's Web services are registered. Each entry in this list contains the G-op-ID of an imported operation, the WS-ID of the service that imported it, and the ID (CA-ID) of the community agent CA_i. To enable fast access to CL_j, we sort it on the G-op-ID column using *Counting Sort* algorithm [6]. RB_j contains a set of rules that enable MA_j to react to changes issued by community providers.

Upon reception of the registration message, CA_i updates the content of C_i's *members list* ML_i (figure 4, steps (c) and (d)). Each entry in ML_i contains the ID of the imported operation (op-ID), the ID of the importing service (*WS-ID*), the ID of *WS-ID*'s agent (*MA-ID*), *WS-ID*'s *status* ("available ", "unavailable", or "unsubscribed"), and the list *NotImp* of *G-op-ID*'s attributes not imported by *WS-ID*. The *NotImp* column is assigned with the content of *NotImpSet* included in the registration message. We sort ML_i on the op-ID column using *Counting Sort* algorithm to enable fast access to ML_i [6].

Figure 5 gives the algorithm executed by a community agent CA_i and member agent MA during registration. SP_Register(WS-ID,name,desc,imported) describes the actions executed by CA_i as a reaction to a registration message sent by MA. The *Insert_Member()* function allows the insertion of a new member in C_i's description. *Insert_ML()* function allows the insertion of a new entry in the ML_i list for members.

```
(00) Member Agent $MA$ Algorithm {
(01) Upon Reception of
(02)  Register (WS-ID,name,desc,imported,$C_i$) statement From SP
(03)  Parse the registration statement;
(04)  Send SP_Register (WS-ID,name,desc,imported) To $CA_i$; }

(00) Community Agent $CA_i$ Algorithm {
(01) Upon Reception of
(02)  SP_Register (WS-ID,name,desc,imported) message From MA-ID
(03)  Insert_Member ($C_i$,WS-ID,name,desc,imported);
(04)  For each G-op-ID $\in$ imported
(05)   Do Insert_ML (G-op-ID,WS-ID,MA-ID,''available'',NotImpSet); }
```

Figure 5. Service registration: Member and community agent algorithms.

4.2. Importing generic operations

Service providers use generic operations as "templates" to define their operations. A Web service may offer all or some of the generic operations defined within a community. The provider specifies the G-op-IDs of the operations imported by its service. By adopting a generic operation, the service provider "promise" to abide by all attributes (syntactic, semantic, and behavioral) of that operation except those changed explicitly during importation. Providers may customize generic operations to best fit their capabilities via *importing statements*. Customization has the important advantage of enabling flexible and personalized Web service descriptions. It is important to note that customization process does not affect the description of generic operations. Two service providers may import the same generic operation in different ways.

We define three importing statements: *projection*, *extension*, *adjustment*. Importing statements are defined within member agents. Projection, extension, and adjustment may be combined to define imported operations. For example, a service provider may use projection and extension to remove existing parameters from a generic operation and add new ones.

Projection. A generic operation G-op-ID imported by *projection* uses a subset of the input/output parameters defined in G-op-ID. The rest of message parameters are not imported by the service and hence, are included in the *NotImpSet* sent by the member agent to the community agent. Assume that a `checkRegistrationAAA` operation includes `register` (boolean) and `registrationDate` (date) as output parameters. A service provider may customize this operation by keeping only the `register` parameter if it is not interested in returning citizen's registration date. In what follows, we give the general form of a *Project* statement. The input and output clauses give the subset of G-op-ID's parameters supported by the imported operation:

Project *G-op-ID*
  Input <*list-of-parameter-names*>
  Output <*list-of-parameter-names*>

The *AttSet* submitted with the registration message takes the form $\{(att, value)\}$ where *att* is a projected parameter and the content of *value* is "null" since no new value is assigned to the input or output parameter. We refer to *AttSet.Attributes* as the list of projected parameters. The list of not imported attributes *NotImpSet* is defined by the expression $(In(op_{ik}) \cap Out(op_{ik})) - AttSet.Attributes$.

Extension. An imported operation defined by *extension* adds input and/or output parameters to the corresponding generic operation. The new parameters and their values are included in the registration messages sent by the member agent to the community agent. For example, a service provider (e.g., a volunteer center) may extend the `mealsOnWheels` operation by adding the deliverer's cell phone number as an output parameter. Service providers must assign values to the attributes of each new message parameters, namely data type, unit, business role, and language. Below is the general form of an extension statement:

Extend *G-op-ID*
  Input [(*<name>, <data-type>, <unit>, <business-role>, <language>*)]*
  Output [(*<name>, <data-type>, <unit>, <business-role>, <language>*)]*

The *AttSet* submitted with the registration message takes the form $\{(att,value)\}$ where *att* is a new attribute added by the extension statement and *value* = (*<name>*, *<data-type>*, *<unit>*, *<business-role>*, <language>). The list of not imported attributes *NotImpSet* is empty since all message parameters are imported.

Adjustment. The aim of an adjustment statement is to modify the content of a generic operation's attributes. Service providers may assign values to attributes whose content is undefined (e.g., qualitative attributes) or change the content of previously assigned attributes (e.g., language attribute). Adjustment is done by adding a value to an attribute (Add clause) or deleting an existing value from it (Delete clause). For example, service providers may modify the language attribute if their operation supports a language different from the one specified by community providers. The Add and Delete clauses may be combined to remove and add values to an attributes. The new and deleted values are included in the registration messages sent by the member agent to the community agent. The general form of an adjustment statement is given below:

Adjust *G-op-ID*
  Add [*<value>* **To** *<attribute>*]*
  Delete [*<value>* **From** *<attribute>*]*

The *AttSet* submitted with the registration message takes the form {(att,["+"/"−"],value)} where *att* is an attribute modified by the adjustment statement. The *value* is preceded by a "+" or "−" symbol depending on whether that value is added or deleted from *att* respectively. The list of not imported attributes *NotImpSet* is empty since all message parameters are imported.

5. A peer-to-peer approach for managing communities

Communities and their members operate in a highly dynamic environment where changes can be launched to adapt to actual business climate (e.g., economic, politic, organizational). Changes are initiated by community or service providers. At the community providers side, generic operations may be dynamically added, deleted, and modified. If a generic operation G-op-ID is deleted or modified, then all members that are supporting G-op-ID should be notified to ensure global consistency. At the service provider side, a Web service may cancel its membership with a community, make its operations temporarily unavailable, or modify the definition of its imported operations. The community provider should in this case be notified to avoid references to inexistent or obsolete imported operations.

In our approach, all changes are introduced through member and community agents. Agents automatically interact with their peers to manage changes. We consider two types of changes based on the party that launched them: community or service providers.

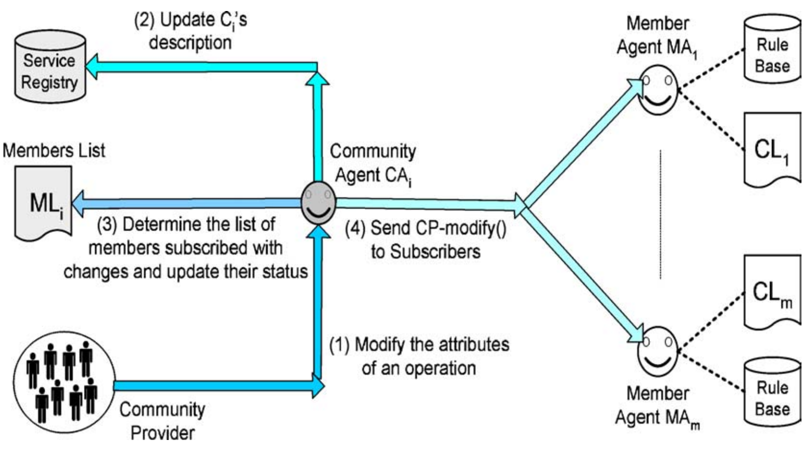
(2) Update C_i's description
Service Registry
Members List
ML_i
(3) Determine the list of members subscribed with changes and update their status
Community Agent CA_i
(4) Send CP-modify() to Subscribers
(1) Modify the attributes of an operation
Community Provider
Member Agent MA_1
Rule Base
CL_1
CL_m
Rule Base
Member Agent MA_m

Figure 6. Propagating changes initiated by a community provider to its members.

5.1. *Propagating changes initiated by community providers*

Community providers (CPs) may modify the definition of their generic operations (figure 6). For example, they may change the pre-operations attribute to reflect new government regulations. For this purpose, each CP$_i$ executes a *Modify* statement defined in its CA$_i$ agent (figure 6, step 1). The statement includes the G-op-ID of the operation to be modified by CP$_i$ and a *ModifySet* that contains the list of attributes to be modified along with their new content. *ModifySet* is defined by the set $\{(<\text{attribute}>,[\text{“}+\text{”}/\text{“}-\text{”}],<\text{value}>)\}$. We use the notation *ModifySet*. *Att* to refer to the set of modified attributes. CA$_i$ will then access the service registry and update C$_i$'s description by changing the content of *ModifySet.Att*'s attributes (figure 6, step 2). In the third step, CA$_i$ accesses ML$_i$ list to determine the list $\mathcal{L}$ of members *subscribed* with CP$_i$'s changes. A member WS-ID is *subscribed* with CP$_i$'s changes if WS-ID imports G-op-ID and at least one attribute in *ModifySet.Att* is imported by WS-ID. CA$_i$ assigns the value “unavailable” to the *status* of each subscribed member. This prevents references to members that imported “obsolete” generic operations (figure 6, step 3). Finally, CA$_i$ sends a *CP-Modify(G-op-ID,WS$_j$,ModifySet)* notification to each subscribed member's MA$_j$ (figure 6, step 4). We give below a formal definition of a subscription:

Definition 4 (Subscription). Let C$_i$ be a community and WS_j be a member that imported an operation op$_{ik}$. WS_j is *subscribed* with changes specified in *CP-Modify(G-op-ID,ModifySet)* if (i) G-op-ID $=$ op$_{ik}$ and (ii) *ModifySet.Att* – *ML$_i$[k].NotImp* $\neq \emptyset$ where k is the entry corresponding to G-op-ID and WS$_j$.

Because pre and post-operations for a given operation op_{ik} may belong to different communities, we associate two “dual” subscription lists PPCS$_i$ and PPMS$_i$ to each community agent CA_i. These lists are used to notify relevant parties (communities and members) about

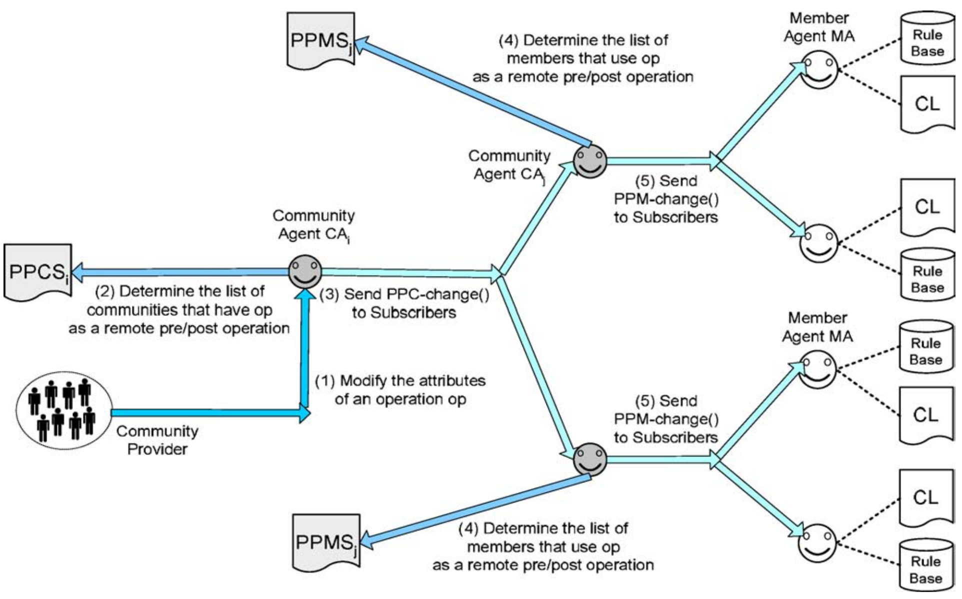
PPMS$_j$
(4) Determine the list of members that use op as a remote pre/post operation
Member Agent MA
Rule Base
CL
Community Agent CA$_j$
(5) Send PPM-change() to Subscribers
Community Agent CA$_i$
PPCS$_i$
(2) Determine the list of communities that have op as a remote pre/post operation
(3) Send PPC-change() to Subscribers
(1) Modify the attributes of an operation op
Community Provider
Member Agent MA
(5) Send PPM-change() to Subscribers
PPMS$_j$
(4) Determine the list of members that use op as a remote pre/post operation

Figure 7. Propagating changes initiated by community providers to their peers.

changes that are related to remote pre and post-operations. PPCS$_i$ (*Pre and Post-operation Community Subscription*) contains the list of communities C_j that use an operation op_{ik} of C_i as a remote pre or post-operation. Each entry in PPCS$_i$ contains the ID (op-ID) of op_{ik} and the ID of C_j's agent (CA-ID). Such entry is created at C_j's definition time; at that time, CA_j sends a *CP-PP-subscribe-CP(op-ID)* to CA_i (CP stands for Community Provider and PP for Pre/Post operation). PPCS$_i$ is used to notify C_j about changes that occur in op_{ik}. Each time CP$_i$ executes a *Modify* statement on op_{ik}, CA_i accesses PPCS$_i$ list and sends a notification *CP-PP-Alert-CP(op_{ik})* to each community agent CA_j that uses op$_{ik}$ as a remote pre or post-operation (figure 7). We sort *PPCS$_i$* on the op-ID column using *Counting Sort* algorithm to enable fast access to the list [6]. Once the *CP-PP-Alert-CP(op_{ik})* message is received by CA_j, CA_j notifies C_j's members using the PPMS$_j$ list.

PPMS$_j$ (*Pre and Post-operation Member Subscription*) contains the list of C_j's members that use the operations op_{ik} of other communities C_i as remote pre or post-operations. This list is created at C_j's definition time. It is used to notify members about changes that occur in their pre and post-operations. Each entry in PPMS$_j$ contains the ID (CA-ID) of the community agent CA_i, the ID (PP-ID) of the remote pre or post-operation op_{ik}, and the ID (MA-ID) of the agent of a C_j's member that uses op_{ik} as a remote pre or post-operation. CA_j accesses the PPMS$_j$ list and sends a notification message *CP-PP-Alert-SP(op_{ik})* (SP stands for Service Provider) to each member agent MA-ID that uses op_{ik} as a remote pre-operation (figure 7). We sort *PPMS$_i$* on the CA-ID column using *Counting Sort* algorithm to enable fast access to the list [6]. To prevent references to members that imported "obsolete" generic operations, CA$_j$ assigns the value "unavailable" to the *status* of MA-ID entries in *ML$_j$* list.

```
(00) Community Agent $CA_i$ Algorithm {
(01) Upon Reception of
(02)  **Modify (G-op-ID,ModifySet) statement From** $CP_i$
(03)   Update_Member ($C_i$,G-op-ID,ModifySet);
(04)   For each $ML_i$[k].op-ID | ($ML_i$[k].op-ID=G-op-ID)$\wedge$(ModifySet-$ML_i$[k].NotImp$\neq\emptyset$)
(05)    Do $ML_i$[k].status = "unavailable";
(06)    Send CP_Modify(G-op-ID,$ML_i$[k].WS-ID,ModifySet) To $ML_i$[k].MA-ID;
(07)   If ModifySet contains pre or post-operation attributes
(08)    Then For each $PPCS_i$[k].op-ID | $PPCS_i$[k].op-ID==G-op-ID
(09)     Do Send CP-PP-Alert-CP(G-op-ID) To $PPCS_i$[k].CA-ID;
(10)  **CP_PP_Alert_CP (G-op-ID) message From** $CA_j$
(11)   For each $PPMS_i$[k] | $PPCS_i$[k].op-ID==G-op-ID
(12)    Do Let p be the entry in $ML_i$ | $ML_i$[p] has G-op-ID as pre/post-operation
(13)     $ML_i$[p].status = "unavailable";
(14)     Send CP-PP-Alert-SP(G-op-ID) To $ML_i$[p].MA-ID;
(15)  **CP_PP_Remove_CP (G-op-ID) message From** $CA_j$
(16)   Let k be | $PPCS_i$[k].op-ID==G-op-ID $\wedge$ $PPCS_i$[k].CA-ID==CA$_j$;
(17)    remove entry k from $PPCS_i$; }
```

Figure 8. Generic operation modification: Community agent algorithm.

We only consider entries in which MA-ID imported operations that have *op*$_{ik}$ as a remote pre or post-operation.

C$_j$'s provider may remove the remote pre/post operation *op*$_{ik}$ from its generic operation definitions. To propagate this change, CA$_j$ removes from PPMS$_j$ the entry that corresponds to *CA*$_i$ and *op*$_{ik}$. CA$_j$ also sends a *CP-PP-Remove-CP(op*$_{ik}$*)* to *CA*$_i$. Upon reception of this message, *CA*$_i$ removes from PPCS$_i$ the entry corresponding to *CA*$_j$ and *op*$_{ik}$. *CA*$_i$ will hence send no *CP-PP-Alert-CP(op*$_{ik}$*)* messages to *CA*$_j$ since C$_j$ does not use op$_{ik}$ as a remote pre or post operation. Figure 8 summarizes the algorithm executed by CA_i for managing the modification of generic operations. The *Update_Member()* function allows the update of the description of a community description within the service registry.

5.2. *Propagating changes initiated by service providers*

The provider of a Web service *WS-ID* may initiate changes that should be sent to their communities (figure 9). A community *C*$_i$ is notified about a change if the change is made on an operation imported from *C*$_i$. All changes are introduced through member agents and automatically forwarded to community agents which reflect those changes at the community level. We define the following service providers changes:

- *Modifying operations. WS-ID*'s provider may modify attributes (e.g., remove a message parameter) of a previously imported operation through modification statements defined in *WS-ID*'s agent *MA* (figure 9, step 1). A modification statement includes the *G-op-ID*

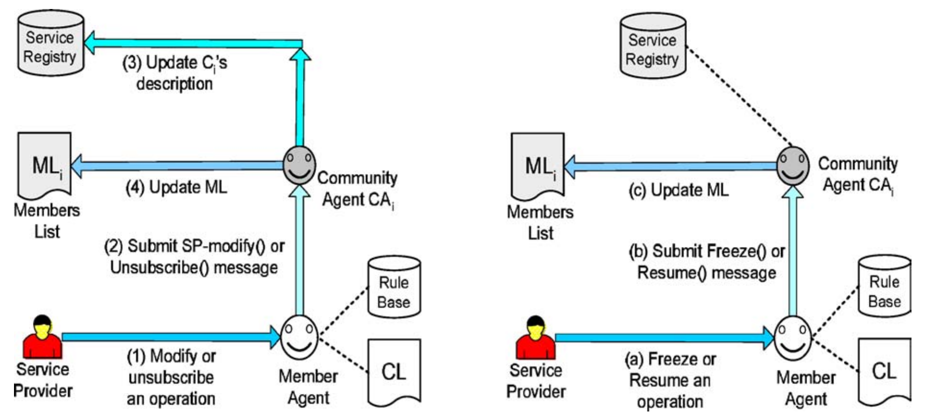

Figure 9. Propagating changes initiated by service providers.

of the operation to be modified, *WS-ID* of the service that imported it, and the importing statement *I-statement* used to do the modification. Once *MA* gets a modification statement, it sends an *SP-Modify(G-op-ID,WS-ID,IType,ModifySet)* message to C_i's agent CA_i (figure 9, step 2). ModifySet is defined as in *CP-Modify()* messages. *IType* gives the type of I-statement ("projection", "extension", "adjust add", and "adjust delete"). Upon reception of the *SP-Modify()* message, CA_i updates C_i's description (figure 9, step 3). If the message concerns a projection statement, CA_i updates the *NotImp* column in the ML_i's entry that correspond to *G-op-ID* and *WS-ID* (figure 9, step 4). It assigns the content of *ModifySet.Att* to this column. If the message concerns an "adjust delete" statement on a remote pre or post operation op_{jl}, then CA_i removes the entry in PPMS_i corresponding to CA_j, op_{jl}, and *WS-ID*'s agent. If the message concerns an "adjust add" statement on a remote pre or post operation op_{jl}, then CA_i adds a new entry in PPMS_i with the values CA_j, op_{jl}, and *WS-ID*'s agent *MA-ID*.

- *Freezing operations.* An imported operation may be "available", "unavailable" (e.g., due to network problem), or "unsubscribed". *WS-ID*'s provider may temporarily make an imported operation *G-op-ID* operation by executing the *Freeze* statement (figure 9, step a) defined in *MA*. The statement includes the G-op-ID of the operation to be frozen and the WS-ID of the service that imported it. As a consequence, *MA* sends an *SP-Freeze(G-op-ID,WS-ID)* message to CA_i (figure 9, step b). CA_i then assigns the "unavailable" value to the *status* column of the ML_i's entry that correspond to *G-op-ID* and *WS-ID* (figure 9, step c).
- *Resuming operations. WS-ID*'s provider may re-activate an operation G-op-ID that has previously been frozen through the *Resume* statement defined in *nMA*. The statement includes a reference to WS-ID and G-op-ID. As a consequence, *MA* sends an *SP-Freeze(G-op-ID,WS-ID)* to G-op-ID's community agent CA_i (figure 9, step b). CA_i then assigns the "available" value to the *status* column of the ML_i 's entry that correspond to *G-op-ID* and *WS-ID* (figure 9, step c).

- *Unsubscribing operations. WS-ID*'s provider may decide not to support an imported operation any more. For that purpose, it executes *MA*'s *Unsubscribe* statement (figure 9, step 1). *MA* informs CA_i about this change (figure 9, step 2) by sending an *Unsubscribe(G-op-ID,WS-ID)* message to it. CA_i then updates C_i's description (figure 9, step 3). It also assigns the "unsubscribed" value to the *status* of the ML_i's entry that correspond to *G-op-ID* and *WS-ID* (figure 9, step 4). CA_i periodically checks the *status* column in ML_i and remove all entries associated with unsubscribed operations.

Figures 10 and 11 depict the algorithm executed by community and member agents for managing changes initiated by service providers. The *Lookup_Community()* function (lines 14) is executed on the communities list *CL*. It returns the CA_i's ID of the community C_i to which the operation G-op-ID belongs.

Each member's agent MA_j also needs to react to change notifications sent by a community agent via *CP-Modify(G-op-ID,WS$_j$,ModifySet)* or *CP-PP-Alert-SP(op$_{ik}$)* messages. The actions to be performed by MA_j as a result of change notification are captured using ECA (*Event Condition Action*) rules [5]. Briefly, the basic semantics of an ECA rule is as follows: when an event occurs, an action is executed if the corresponding condition is true. Event-driven systems are becoming the paradigm of choice for organizing many classes of loosely coupled and dynamic applications. Members react to changes using their own change control policies via local rule specified in their agent. Hence, the reaction to changes can be customized to the peculiarities of each member. Below is an example of ECA rule specified within a member agent:

```
(00) Member Agent $MA$ Algorithm {
(01) Upon Reception of
(02)  Modify (G-op-ID,WS-ID,I-Statement) statement From SP
(03)   Parse the importing statement I-Statement;
(04)   Get ModifySet From I-Statement;
(05)   IType = type of the I-Statement;
(06)   $CA_i$ = Lookup_community($CL$,G-op-ID);
(07)   Send SP_Modify (G-op-ID,WS-ID,IsProjection,ModifySet) To $CA_i$;
(08)  Freeze (G-op-ID,WS-ID) statement From SP
(09)   $CA_i$ = Lookup_community($CL$,G-op-ID);
(10)   Send SP_Freeze (G-op-ID,WS-ID) To $CA_i$;
(11)  Resume (G-op-ID,WS-ID) statement From SP
(12)   $CA_i$ = Lookup_community($CL$,G-op-ID);
(13)   Send SP_Resume (G-op-ID,WS-ID) To $CA_i$;
(14)  Unsubscribe (G-op-ID,WS-ID) statement From SP
(15)   $CA_i$ = Lookup_community($CL$,G-op-ID);
(16)   Send SP_Unsubscribe (G-op-ID,WS-ID) To $CA_i$; }
```

Figure 10. Changes issued by service providers: Member agent algorithm.

```
Community Agent $CA_i$ Algorithm {
Upon Reception of
**SP_Modify (G-op-ID,WS-ID,IType,ModifySet) message From** MA-ID
  Update ($C_i$,G-op-ID,ModifySet);
  Let k be the entry in $ML_i$ | $ML_i$[k].op-ID==G-op-ID $\wedge$ $ML_i$[k].WS-ID==WS-ID;
  Case IType
    "Projection": $ML_i$[k].NotImp = Parameters(G-op-ID) - ModifySet.Att;
    "Adjust Delete": If ModifySet contains pre or post-operations
              Then remove from PPMS$_i$ the entry related to G-op-ID and MA-ID;
    "Adjust Add": If ModifySet contains pre or post-operations
              Then add to PPMS$_i$ an entry for G-op-ID and MA-ID
**SP_Freeze (G-op-ID,WS-ID) message From** MA-ID
  Let k be the entry in $ML_i$ | $ML_i$[k].op-ID==G-op-ID $\wedge$ $ML_i$[k].WS-ID==WS-ID;
  $ML_i$[k].status = "unavailable";
**SP_Resume (G-op-ID,WS-ID) message From** MA-ID
  Let k be the entry in $ML_i$ | $ML_i$[k].op-ID==G-op-ID $\wedge$ $ML_i$[k].WS-ID==WS-ID;
  $ML_i$[k].status = "available";
**SP_Unsubscribe (G-op-ID,WS-ID) message From** MA-ID
  Let k be the entry in $ML_i$ | $ML_i$[k].op-ID==G-op-ID $\wedge$ $ML_i$[k].WS-ID==WS-ID;
  $ML_i$[k].status = "unsubscribed'; }
```

Figure 11. Changes issued by service providers: Community agent algorithm.

Rule R_1
 Event CP-Modify(G-op-ID,WS$_j$,ModifySet)
 Condition Source $= C_1 \wedge$ *pre-operations* $\in$ *Modify.Att*
 Action <notify the service provider to change its internal business logic>;
  send *Resume(G-op-ID,WS$_j$)* message to **source**

R_1 states that whenever the member receives a change notification issued by C_1's provider and if the change concerns the *pre-operations* attribute, the service provider should reflect the change by modifying its internal business logic. The member agent then sends a *Resume(G-op-ID,WS$_j$)* message to C_1's agent. The message is a confirmation that the member has locally reflected the changes done by CP_1. Figure 12 summarizes the change reaction algorithm executed by member agents.

6. Implementation

We provide a prototype implementation of the proposed approach using Web service standards such as SOAP and UDDI (figure 13). Users access the *Semantic Web Service Manager* through a graphical interface (implemented in HTML/Servlet). We identify two types of users: community providers and service providers. All users' requests are received by the *request handler* which forwards them either to the *community builder* or *service registrar* depending on their type.

```
Member Agent $MA$ Algorithm {
Upon Reception of
 CP_Modify (G-op-ID,WS-ID,ModifySet)
  For each rule=(event,condition,actions) in the rule base RB
    | event==CP_Modify(G-op-ID,WS-ID,ModifySet) $\wedge$ condition==.T
    Do Execute actions;
 CP_PP_Alert_SP(G-op-ID) From $CA_i$
  For each rule=(event,condition,actions) in the rule base RB
    | event== CP_PP_Alert_SP(G-op-ID) $\wedge$ condition==.T
    Do Execute actions; }
```

Figure 12. Reaction to changes issued by community providers: Member agent algorithm.

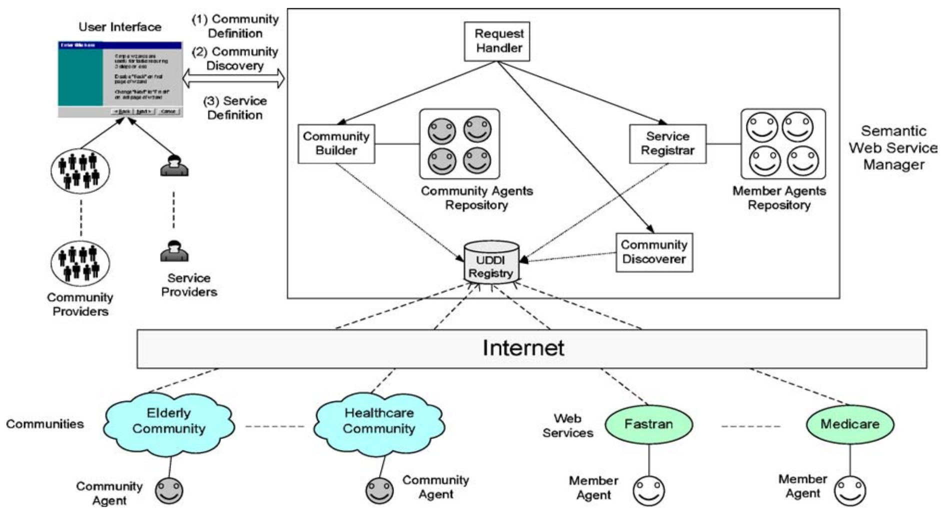

Figure 13. Architecture.

If the request is for creating a community (issued by community providers), it is handled by the community builder. Community providers define a community either by sending an XML document that instantiates the community ontology or by filling out a form via the graphical interface. In both cases, the community builder creates an entry for the community in the UDDI registry. We implement the UDDI with *Systinet's WASP UDDI Standard 3.1*. *Cloudscape 4.0* database is used to create the registry for the UDDI. All communications are encapsulated in SOAP envelopes. *Apache SOAP* provides the tools necessary for deploying SOAP messaging. Every community is represented in the registry by a *tModel* [1]. We have designed specific tModels for communities. The *community tModel* contains a URL pointer to the description of the community. When defining a community, an agent called *community agent* (implemented using IBM Aglets) is downloaded from the *community agents repository* to the community provider's site. All subsequent access (e.g., modify a community) to the community are done through the community agent.

The second type of requests concerns the discovery of communities. These requests are issued by service providers to inquire about communities of interest. They are received by the *community discoverer* which implements *UDDI Inquiry Client* using WASP UDDI API. Once a community is discovered, the service provider may submit a request for registration with this community. Such a request is forwarded to the *service registrar* which creates an entry for the Web service in the UDDI registry. The Web service is represented by a *tModel* in the registry. We have also designed tModels for community members called *member tModel*. As a result of the registration process, a member agent (implemented using IBM Aglets) is downloaded to the service provider's site. The member agent is then responsible for handling all subsequent requests issued by the service provider. The deployment of community and member agents caters for a peer-to-peer and distributed architecture for managing communities and Web services.

7. Related work

Standardization efforts including *UDDI* (Universal Description, Discovery, and Integration) and *WSDL* (Web Services Description Language) are underway to support Web services [1]. The main impediment of current Web service standards is their limited support for semantics. A major effort towards enabling semantic Web services is *DAML-S* (DARPA Agent Markup Language) [10]. *DAML-S* defines a semantic markup of Web services based on ontologies. Several features distinguish the approach proposed in this paper from *DAML-S*. First, *DAML-S* proposes an ontology for Web services *not* for community of Web services. Our approach provides means for the semantic description of Web services and their ontological organization into communities. Second, the static semantics in *DAML-S* mostly focuses on describing operations' features. We define a broader view of static semantics by describing semantics both at the operation and message levels. Third, *DAML-S* gives little support for the dynamic semantics of Web services. It does not allow the specification of pre-operations and post-operation which are particularlyimportant for enabling the automatic generation of business processes. Additionally, the notion of behavior and business logic is not explicitly defined. Fourth, *DAML-S*providers define their service operations from scratch. In our approach, providers inherit the functionalities of a community simply by registering their services with it. They may also personalizethat community to best fit their capabilities.

WSMF (Web Service Modeling Framework) combines the concepts of Web services and ontologies to cater for semantic Web enabled services [4]. *WSMF* is still in its early stage. The techniques for the semantic description of Web services are still ongoing. Furthermore, *WSMF* does not address the issue of organizing semantic Web services.

8. Conclusion

In this paper, we propose an ontological framework to cater for a meaningful organization and description of Web services. We introduce the concept of community to cluster Web services based on their domain of interest. We develop an ontology called *community ontology* that serves as a "template" for describing communities and Web services. The proposed framework lays the foundation for the automatic selection and composition of semantic Web services. We are currently defining a *composability model* that would automatically check whether Web services "can" be combined together. The model would compare different features of Web services including syntactic, semantic (static and dynamic), behavioral, and qualitative properties.

Acknowledgment

The second author's research is partly supported by the National Institutes of Health's NLM Grant 1-R03-LM008140-01.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju, Web Services: Concepts, Architecture, and Applications, Springer Verlag, June 2003, ISBN: 3540440089.
2. T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," Scientific American, May 2001.
3. C. Bussler, B2B Integration, Springer Verlag, May 2003, ISBN: 3540434879.
4. C. Bussler, D. Fensel, and A. Maedche, "A conceptual architecture for semantic Web enabled Web services, SIGMOD Record, vol. 31, no. 4, 2002.
5. C. Collet, T. Coupaye, and T. Svensen, "Naos: Efficient and modular reactive capabilities in an object-oriented database system," in VLDB Conf., Santiago, Chile, September 1994.
6. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C.Stein, Introduction to Algorithms, MIT Press, 2001.
7. D. Fensel, Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce, Springer Verlag, Sept. 2003, ISBN: 3540003029.
8. Object Management Group, Unified Modeling Language Specification. http://www.omg.org/technology/documents/formal/uml.htm, 1999.
9. I. Horrocks, "DAML+OIL: A description logic for the Semantic Web," IEEE Data Engineering Bulletin, March 2002.
10. S.A. Mclraith, T.C. Son, and H. Zeng, "Semantic Web services," IEEE Intelligent Systems, March 2001.
11. C. Petrie and C. Bussler, "Service agents and virtual enterprises: A survey," IEEE Internet Computing, vol. 7, no. 4, 2003.
12. S. Ran, "A model for web services discovery with QoS," SIGecom Exchanges, vol. 4, no. 1, 2003.
13. S. Tsur, S. Abiteboul, R. Agrawal, U. Dayal, J. Klein, and G. Weikum, "Are Web services the next revolution in e-commerce? (Panel)," in VLDB Conference, Sep. 2001.
14. G. Weikum (ed.), Special Issue on Organizing and Discovering the Semantic Web, IEEE Data Engineering Bulletin, March 2002.
15. L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q.Z. Sheng, "Quality driven Web services composition," in Twelfth International World Wide Web Conference, Budapest, Hungary, May 2003.