Using Argumentative Agents to Manage Communities of Web Services

Jamal Bentahar$^{\alpha}$, Zakaria Maamar$^{\beta}$, Djamal Benslimane$^{\gamma}$, and Philippe Thiran$^{\delta}$
$^{\alpha}$CIISE, Concordia University, Canada —— $^{\beta}$CIT, Zayed University, U.A.E
$^{\gamma}$LIRIS, Claude Bernard Lyon 1 University, France —— $^{\delta}$IMRU, University of Namur, Belgium

Abstract— **This paper presents a framework for specifying Web services communities. A Web service is an accessible application that humans, software agents, and other applications in general can discover, compose, and invoke in order to satisfy users' needs like hotel booking. Web services providing the same functionality are gathered into one community, independently of their origins. This framework shows how software agents that are able to argue, negotiate, and reason about Web services can be used to specify these Web services and to manage their respective communities. The use of what we call *argumentative agents* helps Web services in being better organized within communities and in achieving the goals for which they are conceived. The community is led by a master component, which among others attracts new Web services to the community, retains existing Web services in the community, and identifies the Web services in the community that will participate in composite Web services. All these operations are managed by interacting agents through flexible conversations made up by argumentation, persuasion, and negotiation phases called *dialogue games*.**

I. Introduction

Recent years have seen an increasing interest in Web services. The W3C defines a Web service as "*a software application identified by a URI, whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts and supports direct interactions with other software applications using XML-based messages via Internet-based applications*". As the number of Web services continues to grow, the need of composing them to build more complex and complete business applications is widely stressed. To facilitate and improve the process of Web services discovery in an open environment like the Internet, it is suggested to gather Web services with similar functionalities into groups known as communities [1], [8], [10]. Although Web services are intensively investigated, the following community-related issues have not been properly addressed yet by researchers: how to initiate, set up, and specify a community of Web services, is the functionality of a Web service the only factor that drives the establishment of a community, and how to specify and manage the Web services that reside in a community?

It is widely recognized that software agents are a promising technology to develop a new generation of Web-based applications. In fact, agents are associated with a powerful set of metaphors and techniques for designing, implementing, and verifying complex, distributed systems such as electronic trading and distributed business process. Although there is little consensus about the definition of a software agent, it is generally held that agents are autonomous pieces of software, able to take initiative in order to satisfy some goals. Several formal logics have been proposed to specify and implement agents like epistemic and doxastic logic dealing with attitude of knowledge and belief, deontic logic treating obligations, BDI logic for beliefs, desires, and intentions, KARO logic specifying knowledge, beliefs, actions, and abilities, and last but not least commitment and argument logic dealing with social commitments and arguments ([11] for an overview).

The advantages of using agent technology to develop Web services have already been identified. In [6], Li et al. propose a framework based on agent-oriented interaction in order to develop dynamic service-oriented operations. The idea is to model and implement service functionalities with interacting autonomous agents. In [4], Dale and his colleagues develop an evening organizer by combining Web services and agents. However, using *argumentative agents* with logic-based reasoning capabilities to develop Web services and specially Web services communities has not been exploited yet. Simply put an argumentative agent complies with a dialectical process when it aims at affirming or disavowing conclusions to convey to peers. The purpose of this paper is to address this challenging issue by proposing a formal and computational framework for specifying and managing Web services communities using argumentative agents. The idea is to allow these communities to be self-managed: their Web services associated with argumentation-based agents make decisions and engage in complex and flexible conversations.

In Section II, we present the architecture of Web services communities and we discuss the operation of such communities. In Section III, we specify the argumentative agents that specify Web services and manage their respective communities. In Section IV, we present the argumentative agent-based framework for Web services communities. In Section V, we conclude and identify some directions for future work.

II. Web Services Communities

A. Definition and Architecture

In Longman Dictionary, community is "*a group of people living together and/or united by shared interests, religion, nationality, etc*". When it comes to Web services, Benatallah et al. define community as a collection of Web services with a common functionality, although these Web services have distinct non-functional properties like different providers and different QoS parameters [1]. Medjahed and Bouguettaya use community to cater for an ontological organization of

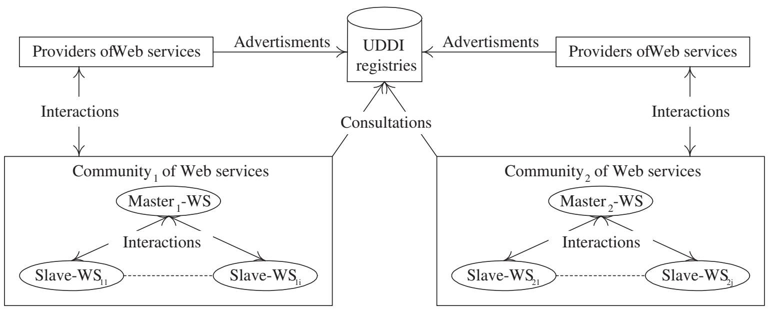

Fig. 1. Architecture of an environment consisting of several Web service communities

Web services that share the same domain of interest [10]. Our definition geos beyond gathering similar Web services in a community and considers a community as a means for providing a common description of a desired functionality (e.g., `FlightBooking`) without explicitly referring to any concrete Web service (e.g., `EKFlightBooking`) that will implement this functionality at run-time [8].

Fig. 1 represents the architecture we developed to manage communities of Web services. The components of this architecture include providers of Web services, UDDI registries, and communities. A community is dynamic by nature. It is established and dismantled according to specific scenarios and protocols, which we discuss in Section II-B. UDDI registries receive advertisements of Web services from providers. Several UDDI registries could be made available to providers during advertisement, but this is outside this paper's scope.

Two communities of Web services are shown in Fig. 1. They could for example offer `AirfareQuotation` and `HotelBooking` functionalities, respectively. A master component always leads a community. The master component could itself be implemented as a Web service for compatibility purposes with the rest of the Web services in the community. These Web services are now denoted as slaves and have in common the functionality that labels the community in which they run. Within the same community, slave Web-services compete to participate in composite Web services since they all offer the same functionality but in a different set-up.

Each Web service is associated with an argumentative agent able to reason and interact with other agents in the community (Section III). This allows Web services to engage in conversations with each other. Persuasions and negotiations are two examples of conversations that argumentative agents are able to initiate. Such conversations are of great importance in a Web services community. The agent-based master Web-service should now be able to persuade a new Web service to join or remain in its community. Furthermore, within the same community, agent-based slave Web-services should now be able to negotiate together their participation in composite Web services. All these communications are regulated by a set of conversation policies that are detailed in Section IV.

One of the responsibilities of the master Web-service is to attract Web services to sign up in the community it heads using multiple types of rewards (Section II-B). As a result, the master Web-service interacts with the UDDI registries on a regular basis, so it is kept informed regarding the latest changes in the content of these UDDI registries. These changes concern the advertisements of Web services. Furthermore, a new Web service can contact the master Web-service of a given community if it is interested in being part of this community. An additional responsibility of the master Web-service in a community is to nominate the slave Web-service that will participate as component in a composite Web service. To this end and as suggested in [8], the master uses the contract-net protocol [14] by sending a call for bids out to all the slave Web-services. The call for bids always comes along with the non-functional criteria that the user sets for selecting Web services like response time and execution cost. Prior to getting back to the master Web-service, the slave Web-services assess their status [7] and check their capacities of meeting these criteria. Only the slave Web-services that are interested in bidding get back to the master Web-service. This latter screens all the bids before choosing the best one, e.g., a slave Web-service's execution cost and reliability meeting the user's requirements. The winning, slave Web-service is then notified so, it can get ready for execution when requested. The rest of the slave Web-services that expressed interest but were not selected, are notified as well. If several slave Web-services are selected because they offered the same bid, the master Web-service informs them that they should negotiate among themselves and persuade each other to select just one.

In a Web services community, the master Web-service is designated in two different ways. The first way, which we adopt in our work, is to have a dedicated Web service that will play the master role during the time being of a community. It is understood that the leader Web-service never participates in compositions. The second way of designating a master Web-service is to identify a slave Web-service from the list of slave Web-services that already populate a community. This identification could happen on a voluntary basis or after running election between the slave Web-services. To keep the paper self-contained, additional details on the way a master Web-service is designed, are not provided.

B. Operation

The operation of a community of Web services revolves around the following questions: how to develop/dismantle a new/existing community, how to attract new Web services to be enrolled in an existing community, and how to retain existing Web services in a community? All these activities are regulated through argumentative conversations between agent-based Web services.

1) Community of Web services development: A community is primarily established to gather the Web services with the same functionality. This is a designer-driven activity and occurs in two steps. The first step is to define the functionality, e.g., `FlightBooking`, of the community by binding to a specific ontology. This binding is crucial since providers of Web services use different terminologies to describe the functionality of their respective Web services. For example, `FlightBooking`, `FlightReservation`, and `AirTicketBooking` are all about the same functionality. The description of a Web service's functionality needs to be mapped onto the description of the functionality of the community using a specific ontology (i.e., ontology consists of concepts, axioms, relations, and instances).

The second step in developing a community is to deploy the master Web-service that leads the community and takes over the multiple responsibilities we listed in Section II-A. One of these responsibilities is to invite Web services to sign up in the community of this master Web-service by using different types of rewards. Another responsibility is to check the credentials of a Web service before this latter gets admitted in the community. The credentials could be related to QoS, protection mechanisms, interaction protocols, etc. Credential checking can boost the security level within a community as well as enhance the trustworthiness level of a master Web-service towards the slave Web-services of its community.

Dismantling a community of Web service is also a designer-driven activity that happens upon request from the master Web-service. If this latter notices that the number of Web services in the community is less than a certain threshold and the number of participation requests in composite Web services that arrive from users over a certain period of time is less than another threshold, the community could be dismantled. Both thresholds are set by the designer. A slave Web-service that is ejected from a community is invited to join other communities subject to similarity assessment of their respective functionalities.

2) Web services attraction and retention: Attracting new Web services to a community and retaining the existing Web services in a community fall under the responsibilities of the master Web-service. We discussed how a community of Web services could disappear if the number of Web services in this community drops below a certain threshold. On one hand, attracting Web services drives the master Web-service to regularly consult the different UDDI registries looking for new Web services. These latter could recently have been posted on an UDDI registry or have seen their description changed. Changes in a Web service's description raise challenges at the community level since a Web service may no longer be appropriate for a community. As a result, the Web service is invited to leave the community. When a candidate Web service is identified in an UDDI registry according to its functionality, the master Web-service engages in interaction with it. The purpose is to persuade the candidate Web service to register with the community. An argument that is used during this interaction is the high rate of participation of the existing Web services in composite Web services, which is a good indicator of the visibility of a community to the external environment. Other arguments include the short response-time in handling user requests and the efficiency of the security mechanisms against malicious Web services.

Retaining Web services in a community for a long period of time is a good indicator of the following elements:

- Although the Web services in a community are in competition, they expose a cooperative attitude. For instance, Web services are not subject to attacks from peers in the community. This backs the security argument the master Web-service uses to attract new Web services.
- A Web service is to a certain extent satisfied with its participation rate in composite Web services. This satisfaction rate is set by the provider of the Web service. In addition, this is inline with the participation-rate argument the master Web-service uses to attract new Web services.
- Web services know peers in the community that could replace them in case of failure, with less impact on the composite Web services in which they now participate.

Web services attraction and retention shed the light on a third scenario, which concerns Web services being asked to leave a community. A master Web-service could issue such a request upon assessment of the following criteria:

- The Web service has a new functionality, which does not perfectly match the functionality of the community.
- The Web service is unreliable. In different occasions, the Web service failed to participate in composite Web services due to recurrent operation problems.
- The credentials of the Web service were ”beefed up” to enhance its participation opportunities in compositions. It is reported that a Web service may not always fulfill its advertised QoS parameters due to various fluctuations like network status or resource availability [12]. Therefore, some differences between advertised QoS and delivered QoS values occur. However, large differences indicate that the Web service is suffering a performance degradation in delivering its functionalities.

III. Argumentative Agents - Overview

In artificial intelligence, argumentation can be defined as a dialectical process for the interaction of different arguments for and against some conclusions [5], [13]. Agents can be assisted by argumentation to reach a decision and to inform, convince, or negotiate with peers. A single agent may use argumentation techniques to perform its reasoning because it needs to make decisions in highly dynamic environments, considering interacting preferences and utilities. In addition, argumentation can help multiple agents interact rationally, by giving and receiving reasons for conclusions and decisions, within an enriching dialectical process that aims at reaching mutually agreeable joint decisions. Argumentation-based reasoning is an advanced type of reasoning that is more efficient than classical reasoning based on deduction or abduction. In particular, argumentation can serve during negotiation where agents can establish a common knowledge of each other's commitments, find compromises, and persuade one another to make commitments.

Several argumentation theories and frameworks are proposed in the literature ([3], [13]). An argumentation system essentially consists of a logical language $\pounds$, a definition of the *argument concept*, and a definition of the *attack relation* between arguments. The use of a logical language enables argumentative agents to use a logic-based reasoning to effectively reason about arguments in terms of inferring and justifying conclusions, and attacking and defending arguments. Hereafter we define the concepts that will be used in the framework for managing communities of Web services ($\vdash$ stands for classical inference):

Definition 1 (Argument): Let Γ be a knowledge base with no deductive closure. An argument is a pair (H, h) where h is a formula of $\pounds$ and H a subset of Γ such that: (i) H is consistent, (ii) $H \vdash h$, and (iii) H is minimal, so that no subset of H satisfying both (i) and (ii) exists. H is called the support of the argument and h its conclusion.

Definition 2 (Attack): It is a binary relation between arguments. Let (H, h) and (H', h') be two arguments. (H', h') $attacks$ (H, h) iff $H' \vdash \neg h$. In other words, an argument is attacked if and only if there exists an argument that negates its conclusion.

IV. Argumentative Agents for Web Services

A. Formal Foundation

The characteristics of argumentation-based agents discussed in Section III make these agents suitable for modeling Web services in order to make these latter able to interact with each other before joining a community and during their stay in a community. In our framework, argumentative agents act as Web service representatives, reasoning on their behalf and seeking scenarios which maximize their profit (e.g., participation-rate increase). Metadata on contents and features of Web services are represented within the state of the agents. An agent society is a collection of agents whose interactions are regulated by social orders and contracts specifying the agent roles and commitments in the society, as well as the modalities of their interactions. Web services communities are specified as societies of agents connected through a communication network in order to share skills to achieve some overall objectives. The problem of participating in a community and in a composite Web service within a given community then becomes the problem of persuasion and negotiation within agent societies.

Each Web service is associated with an argumentative agent. To be able to reason about Web services and communities, argumentative agents are equipped with knowledge, beliefs, and argumentation capabilities. An agent of a Web service Ag_{WS} knows all details on this Web service in terms of functionality, QoS, etc. The knowledge base of Ag_{WS} is denoted by $KB(Ag_{WS})$. An agent can also have beliefs on other Web services whether in the same community or in other communities. Descriptions of these Web services, their functionalities, QoS, and trust are examples of these beliefs. As explained in the previous Section, the agent's argumentation system is built upon the agent's beliefs and knowledge.

To be able to persuade a Web service to join a community, and to negotiate its participation in a given composite Web service along with the outcome of the contract-net protocol, the master and slave Web-services use persuasion and negotiation techniques based upon their argumentation abilities. Hereafter, we specify a logic-based persuasion and negotiation protocol argumentative agent-based Web services use. This protocol is specified as a combination of a set of initiative/reactive *dialogue games*. Dialogue games can be thought of as interaction games in which each agent plays a move in turn by performing utterances according to a predefined set of rules [9]. Dialogue games have the advantage of being more flexible than classical protocols such as FIPA-ACL protocols. This is because a dialogue game can be specified as a combination of small conversation policies that agents can combine by reasoning about them using a set of logical rules [2]. From a logical point of view, the game moves are considered as communicative actions that agents perform by producing utterances and arguments. These actions are *Assert*, *Accept*, *Refuse*, *Challenge*, *Justify*, *Attack*, and *Defend*. It should be noted that *attacks* is the relation between arguments, and *Attack* is the communicative actions.

Definition 3 (Conversation Policy): Let $Action_{Ag_{WS_1}}$ and $Action_{Ag_{WS_2}}$ be two communicative actions performed by Ag_{WS_1} and Ag_{WS_2} respectively, and let $Cond$ be a formula from the logical language $\pounds$. A conversation policy for an agent-based Web service is a logical rule indicating that if Ag_{WS_1} performs $Action_{Ag_{WS_1}}$, and that $Cond$ is satisfied, then Ag_{WS_2} will perform $Action_{Ag_{WS_2}}$ afterwards. This rule is expressed as follows:

$$Action_{Ag_{WS_1}} \xrightarrow{Cond} Action_{Ag_{WS_2}}$$

$Cond$ is expressed in terms of the possibility of generating an argument from an agent's argumentation system. We distinguish between two types of arguments: private arguments that an agent manages and does not reveal, and public arguments that an agent uses during conversation. We introduce the following sets:

$$PrSupport(Ag_{WS}, p) = \{p'/p' \vdash p\}$$
$$PbSupport(Ag_{WS}, p) = \{Insert(Ag_{WS}, q)/q \vdash p\}$$

$PrSupport(Ag_{WS}, p)$ is the set of Ag_{WS}'s private arguments supporting the proposition p. $PbSupport(Ag_{WS}, p)$ is the set of commitments created by Ag_{WS} to support the proposition p. This set is closed under the support relation, i.e.,

$$p_2 \in PbSupport(Ag_{WS}, p_1) \wedge p_1 \in$$
$$PbSupport(Ag_{WS}, p_0) \Rightarrow p_2 \in Pb_Support(Ag_{WS}, p_0)$$

$p \lhd Arg\_Sys(Ag_{WS})$ formula expressed in $\pounds$ denotes the fact that a propositional formula p can be generated from the

Ag_{WS}'s argumentation system denoted by $Arg\_Sys(Ag_{WS})$. The formula $\neg(p \lhd Arg\_Sys(Ag_{WS}))$ indicates the fact that p cannot be generated from Ag_{WS}'s argumentation system. A propositional formula p can be generated from an agent's argumentation system, if this agent can find an argument supporting p.

B. Dialogue Games for Web Services Communities

For agent-based Web services we distinguish three types of dialogue games: *entry game*, *chaining games*, and *termination game*. The entry game enables the conversation opening and setting up. The chaining games make it possible to continue the conversation by combining several dialogue games. The conversation terminates when the exit conditions are satisfied (termination game).

The entry game allows Web services to initiate conversations. For example, a master Web-service can invite a new Web service registered in a given UDDI to engage in conversation. If the new Web service accepts, then the master can now start another conversation for the sake of persuading this new Web service to join the community. Within a same community, a Web service can invite other Web services to negotiate about participating in a composite Web service. This occurs, as mentioned in Section II, when several agents provide the same "winning" bid after the master Web-service's call. The master selects one of the winnings to invite the others to a negotiation. We specify the entry game as follows:

$$Assert(Ag_{WS_1}, p) \xrightarrow{C_1} Accept(Ag_{WS_2}, p)$$
$$Assert(Ag_{WS_1}, p) \xrightarrow{C_2} Refuse(Ag_{WS_2}, p)$$

where:

$$C_1 = (p \lhd Arg\_Sys(Ag_{WS_2})) \vee \neg(\neg p \lhd Arg\_Sys(Ag_{WS_2}))$$
$$C_2 = \neg p \lhd Arg\_Sys(Ag_{WS_2})$$

Proposition p is expressed in the logical language $\pounds$ using a shared ontology. This proposition indicates an invitation to start a conversation. If the invited Web service has an argument in favor of p or does not have any argument against p, it accepts the invitation, otherwise, it refuses. For example, if a new Web service is not interested in joining a community due to previous unsuccessful experiences in this community, a refusal is sent to this master Web-service. If a Web service does not have any information about the community, or believes that the community's configuration is efficient, it accepts the invitation.

An important dialogue game in persuasion/negotiation interactions is the *defense game*, which is a part of the chaining games. A Web service adopts this game in order to defend its proposition or offer. For example, a master Web-service defends its invitation to a new Web service with various arguments like participation rate of the existing slave Web-services in composite Web services, community's efficient configuration, and why new Web services are needed. We specify the defense game as follows:

$$Defend(Ag_{WS_1}, (H, h)) \xrightarrow{C_1} Accept(Ag_{WS_2}, h)$$
$$Defend(Ag_{WS_1}, (H, h)) \xrightarrow{C_2} Challenge(Ag_{WS_2}, H)$$
$$Defend(Ag_{WS_1}, (H, h)) \xrightarrow{C_3} Attack(Ag_{WS_2}, (H', h'))$$

where:

$$C_1 = H \lhd Arg\_Sys(Ag_{WS_2})$$
$$C_2 = \neg(H \lhd Arg\_Sys(Ag_{WS_2})) \wedge \neg(\neg H \lhd Arg\_Sys(Ag_{WS_2}))$$
$$C_3 = (H' \lhd Arg\_Sys(Ag_{WS_2})) \wedge (H', h')\ attacks\ (H, h)$$

The generation of a set of formulae H from Ag_{WS_2} is defined as follows:

$$H \lhd Arg\_Sys(Ag_{WS_2}) \stackrel{\triangle}{=} \forall h_i \in H\ h_i \lhd Arg\_Sys(Ag_{WS_2})$$

By definition, $Defend(Ag_{WS_1}, (H, h))$ means that Ag_{WS_1} asserts argument (H, h) to defend proposition or offer h, and $Attack(Ag_{WS_2}, (H', h'))$ means that Ag_{WS_2} asserts argument (H', h') to attack argument (H, h). Ag_{WS_2} accepts Ag_{WS_1}' argument if it can generate this argument from its knowledge base. If Ag_{WS_2} can not generate any argument for or against this argument, it challenges the argument by asking for explanations (*challenge game*). Finally, Ag_{WS_2} attacks the argument if it can generate an attacker argument (*attack game*). The challenge game is specified as follows:

$$Challenge(Ag_{WS_1}, H) \xrightarrow{C_1} Justify(Ag_{WS_2}, (H', H))$$

where:

$$C_1 = PrSupport(Ag_{WS_2}, H)$$

Condition C_1 should always be true since a Web service must always be able to justify its propositions and assertions (*justification game*). We specify the justification game as follows:

$$Justify(Ag_{WS_1}, (H, h)) \xrightarrow{C_1} Accept(Ag_{WS_2}, h)$$
$$Justify(Ag_{WS_1}, (H, h)) \xrightarrow{C_2} Challenge(Ag_{WS_2}, H)$$
$$Justify(Ag_{WS_1}, (H, h)) \xrightarrow{C_3} Attack(Ag_{WS_2}, (H', h'))$$

Similar to the defense game, in the justification game, Ag_{WS_2} can either accept, challenge, or attack Ag_{WS_1}'s justification. C_1, C_2, and C_3 are identical to the conditions in the defense game.

Finally, the attack game is specified as follows:

$$Attack(Ag_{WS_1}, (H, h)) \xrightarrow{C_1} Accept(Ag_{WS_2}, h)$$
$$Attack(Ag_{WS_1}, (H, h)) \xrightarrow{C_2} Challenge(Ag_{WS_2}, H)$$
$$Attack(Ag_{WS_1}, (H, h)) \xrightarrow{C_3} Attack(Ag_{WS_2}, (H', h'))$$
$$Attack(Ag_{WS_1}, (H, h)) \xrightarrow{C_4} Refuse(Ag_{WS_2}, h)$$

where:

$$C_1 = H \lhd Arg\_Sys(Ag_{WS_2})$$
$$C_2 = \neg(H \lhd Arg\_Sys(Ag_{WS_2})) \wedge \neg(\neg H \lhd Arg\_Sys(Ag_{WS_2}))$$
$$C_3 = (H' \lhd Arg\_Sys(Ag_{WS_2})) \wedge ((H', h')\ attacks\ (H, h))$$
$$C_4 = \neg h \in KB(Ag_{WS_2}))$$

An agent-based Web service Ag_{WS_2} accepts an attacker's argument if it can generate a support from its argumentation system. If it cannot generate this support nor negate it, the agent challenges it. If it can generate a counter-attacker argument, then it plays the attack game. Otherwise, it refuses the attacker's argument. This *refuse move* can be played if the negation of the attacker's argument conclusion is in Ag_{WS_2}'s knowledge base. We note that in this case Ag_{WS_2} cannot play the attack game since it does not have a counter-argument but only a knowledge about the negation of the argument conclusion. This possibility of refusal move makes defense game and attack game two different games. The defense game is the first one after accepting to engage in persuasion/negotiation conversation. Consequently, the invited Web service can attack the conclusion of the inviter.

Having specified the different dialogue games agent-based Web services can use in their interactions to manage their communities, we need to specify how these games could be combined. We notice that during the same conversation, an agent-based Web service cannot play the same move more than once. For example, if it uses an argument, it cannot use it afterwards during this conversation (reiterations are prohibited). We also notice that dialogue games can be played sequentially or in parallel. For example, a Web service can accept a part of the argument presented by another Web service and challenge a second part in parallel. Playing games in parallel makes Web services conversations more flexible. The conversation terminates (termination game) either by a final acceptance or by a refusal. There is a final acceptance when a Web service accepts the initial proposition (for example accept to join the community) or when an agreement is achieved. The persuasion/negotiation protocol for agent-based Web services (*PNAWS protocol*) that combines these games can be described using the BNF grammar. To this purpose, we first introduce the following definitions where G_1, G_2, and G_3 be three dialogue games:

$$\begin{aligned}
G_1 \ //_{\geq 1}\ G_2 &\triangleq\ G_1 \mid G_2 \mid G_1 \ //\ G_2 \\
G_1 \ //_{opt}\ G_2 &\triangleq\ \epsilon \mid G_1 \ //_{\geq 1}\ G_2 \\
//(G_1 \ ;\ G_2 \ ;\ G_3) &\triangleq\ (G_1 \ //_{\geq 1}\ G_2) \ //_{opt}\ G_3 \\
&\quad \mid (G_1 \ //_{opt}\ G_2) \ //_{\geq 1}\ G_3
\end{aligned}$$

ϵ is the empty dialogue game, "$|$" is the choice symbol, "$;$" is the sequence symbol, and "$//$" is the parallelization symbol. $G_1 \ //\ G_2$ means that an agent can play two games in parallel. Assuming that the entry game is successful(accepted), our *PNAWS protocol* can be defined as follows:

$$\begin{aligned}
PNAWS &= \ \mathit{entry\ game} \ ;\ \mathit{defense\ game} \ ;\ WSDG \\
WSDG &= \ //(\mathit{acceptance\ move} \ ;\ Ch \ ;\ \mathit{Att}) \\
Ch &= \ \mathit{challenge\ game} \ ;\ \mathit{justification\ game} \\
&\quad ;\ (WSDG \mid Refusal) \\
\mathit{Att} &= \ \mathit{attack\ game} \ ;\ (WSDG \mid Refusal)
\end{aligned}$$

V. Discussion and Future Work

In this paper, we presented a framework to manage Web services residing in communities. A community permits gathering Web services with similar functionalities. We addressed several aspects related to the specification and management of a community such as establishing and dismantling a new or existing community, attracting new Web services to join an existing community, retaining existing Web services in a community, and regulating the interactions between Web services using argumentation-based dialogue games. Web services are specified as argumentative agents equipped with beliefs, knowledge, and logical reasoning capabilities. The use of such agents makes Web services autonomous and helps them better manage their roles in a community.

As future work, we plan to look into the computational complexity of this framework. Although we know that logic-based reasoning is generally "hard", using a simple logic like *Horn* logic will make this reasoning tractable. For Web services, this logic can be successfully used for specifying the different cases. Furthermore, reasoning about our simple dialogue games is easy to manage. Consequently, this will not result in severe QoS, but in better reliability level. Addressing the security issues of Web service communities is another direction for future work. We plan to investigate trust networks, role-based trust management, and trust negotiation. In addition, we are looking into various aspects such as specifying a community from three different dimensions: functional (what needs to be done to advertise the functionality of a community), behavioral (what needs to be done to achieve the functionality of a community), and information (what needs to be provided to the functionality of a community).

References

[1] B. Benatallah, Q. Z. Sheng, and M. Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7(1), January/February 2003.

[2] J. Bentahar. *A Pragmatic and Semantic Unified Framework for Agent communication*. PhD thesis, Laval University, Department of Computer Science & Software Engineering, 2005.

[3] C.I. Chesñevar, A. Maguitman, and R. Loui. Logical Models of Argument. *ACM Computing Surveys*, 32, 2000.

[4] J. Dale, L. Ceccaroni, Y. Zou, and A. Agam. Implementing Agent-based Web Services. In *Proceedings of The AAMAS'03 Workshop on Challenges in Open Agent Systems*, Melbourne, Australia, 2003.

[5] M. Elvang-Goransson, J. Fox, and P. Krause. Dialectic Reasoning with Inconsistent Information. In *Proceedings of The 9th Conference on Uncertainty in Artificial Intelligence (UAI'1993)*, Washington, DC, US, 1993.

[6] Y. Li, W. Shen, and H. Chenniwa. Agent-based Web Services Framework and Development Environment. *Computational Intelligence*, 20(4), 2004.

[7] Z. Maamar, D. Benslimane, and N. C. Narendra. What Can Context do for Web Services? *Communications of the ACM*, 49(12), December 2006.

[8] Z. Maamar, M. Lahkim, D. Benslimane, P. Thiran, and S. Sattanathan. Web Services Communities - Concepts & Operations -. In *Proceedings of The 3rd International Conference on Web Information Systems and Technologies (WEBIST'2007)*, Barcelona, Spain, 2007.

[9] P. McBurney and S. Parsons. Games that Agents Play: A Formal Framework for Dialogues between Autonomous Agents. *Journal of Logic, Language, and Information*, 11(3), 2002.

[10] B. Medjahed and A. Bouguettaya. A Dynamic Foundational Architecture for Semantic Web Services. *Distributed and Parallel Databases, Kluwer Academic Publishers*, 17(2), March 2005.

[11] J-J. Meyer and F. Veltman. Inteligent Agents and Common Sense Reasoning. *Studies In Logic and Practical Reasoning. Handbook of Modal Logic - P. Blackburn et al. (Editors)*, 3, 2007.

[12] M. Ouzzani and A. Bouguettaya. Efficient Access to Web Services. *IEEE Internet Computing*, 8(2), March/April 2004.

[13] H. Prakken and G. Vreeswijk. Logics for defeasible argumentation. *Handbook of Philosophical Logic (Second Edition)*, 2000.

[14] R. Smith. The Contract Net Protocol: High Level Communication and Control in Distributed Problem Solver. *IEEE Transactions on Computers*, 29, 1980.