Computing Pragmatic Similarity in Distributed Interaction Systems

Maricela Bravo

Morelos State Polytechnic University, Cuauhnáhuac 566, Texcal,
Morelos, México, CP 62550
mbravo@upemor.edu.mx

Abstract. A Distributed Interaction System (DIS) consists of a set of autonomous software agents, capable of communicating among each other with cooperation and coordination purposes. To achieve their goals, all software agents in a DIS must exchange messages following a protocol. Currently there are research efforts to provide standard mechanisms to achieve cooperation between multiple heterogeneous DIS. However, there are still some issues that must be solved in order to fully automate integration and inter-operation between them. One of this problems is interoperation In this paper we present a novel approach for computing pragmatic similarity for multiple DIS, our objective is to support system developers and integrators with specific information concerning the pragmatic similarity among multiple software agents. We describe a case study to show the applicability of our approach.

Keywords: Distributed Interaction Systems, Pragmatic Similarity Measure.

1 Introduction

A DIS consists of a set of autonomous, independently developed software agents, which have communication capabilities for cooperation and coordination among them. To achieve their goals all software entities in a DIS, exchange messages following an interaction protocol. Currently Internet-based environments have gained more atention than ever, and many software entities independently developed are being deployed on the Web, causing the problem of heterogeneity. Heterogeneity in DIS has many causes: use of different hardware and operating systems, use of different programming languages for implementation, different data base management software, different naming techniques, different data and code formats, etc. Considering this inherent DIS heterogeneity, to provide software entities with mechanisms to be adaptable at run time to interoperate in dynamic and complex environments such as Internet, represents an open problem. Therefore, research efforts to model interactions between different software entities, and evaluate the behavior similarity among them, will benefit the design selection of an interoperable solution for DIS over Internet.

Many authors have approached this problem from a data-based perspective, which we will refer as the data approach. We consider this is a good solution approach,

Advances in Computer Science and Artificial Intelligence.
Research in Computing Science 39, 2008, pp. 155-170

Received 05/07/08
Accepted 01/08/08
Final version 14/09/08

which has proven good results. However, this data approach has been mainly used for heterogeneous information sources integration: such as distributed data bases, vocabularies, ontologies, taxonomies, etc. This data approach would be enough if Internet was populated only with information sources, but it is not. Indeed, Internet has more than only data; it has programs, agents, Web applications, etc. We will refer to this kind of sources as interactive software entities. These interactive software entities have to communicate to each other through the exchange of messages, following interaction protocols or rules.

In this paper we focus on the pragmatic aspect of interactive software entities, in particular we propose an approach for computing pragmatic similarity between software entities, which use different interaction protocols. Our approach is based on the analysis of transition functions extracted from interaction protocols modeled with Finite State Machines (FSM). The aim of this research is to provide software programmers or system integrators with mechanisms to compute pragmatic similarity among interactive software entities, in order to identify or propose a good solution for dynamic interoperation.

To evaluate our solution we have implemented a Web-based interaction environment, into which different software entities can be deployed to execute communications among them. Our environment incorporates an Ontology-based translator to help software entities whenever there is a misunderstanding. We also proposed a pragmatic measure to evaluate a priori the level of similarity. We executed a series of experiments to show that the resulting set of pragmatic relations improve interactions between heterogeneous software entities.

The rest of the paper is organized as follows. In section two we present a brief description of representation formalisms which have been used for modeling, simulating and implementing interactions in DIS. In section three we describe the pragmatic representation of software entities. In section four we present an example to show the applicability of our approach. In section five we describe the implemented interaction environment. In section six we present the experimental results. In section seven we present related work and finally, in section eight we conclude.

2 Representation Formalisms

There are various formalisms reported in literature to model DIS, for example Petri Nets, Colored Petri Nets, Pi-calculus, AUML, BPEL, OWL-S.

Petri Nets. Petri Nets were first introduced by Carl Adam Petri, as a result of his Ph.D. thesis in 1962 [1]. Petri Nets have been used to analyze and verify systems in different areas of science, such as artificial intelligence, concurrent systems, control systems, analysis of networks, etc. Petri Nets represent a traditional formalism for modeling interactions and concurrency. A Petri Net is a directed, connected, bipartite graph with annotations, in which each node is either a place or a transition. Tokens are in places, when there is at least one token in every place connected to a transition, then that transition is enabled.

Colored Petri Nets. Colored Petri Nets [2] are based on Petri Nets, but the have added properties. Tokens are not simply blank markers, but have data associated to

them. A color in a token represents a schema or type specification. Places are sets of tuples, called multi-sets. Arcs specify the schema they carry, and can also specify Boolean conditions. Arcs exiting and entering a place may have an associated function which determines what multi-set elements are to be removed or deposited. Boolean expressions, called guards, are associated with the transitions, and enforce some constraints on tuple elements. Colored Petri Nets are equivalent to Petri Nets, but the richer notation of colored Petri Nets makes them more suitable for modeling interactions with more information.

Pi-Calculus. Is a process algebra presented in [3]. It is a formalism for modeling concurrent processes, whose configurations may change as the process executes over time. In Pi-Calculus the fundamental unit of computation is the transfer of a communication link between two processes. The simplicity of the Pi-Calculus is because it includes only two kinds of entities: names and processes. These entities are sufficient to define interaction behavior.

AUML. Agent Unified Modeling Language is a representation formalism which facilitates the visual development of multi-agent systems, with emphasis on agent conversations. It was first introduced by [4]. AUML is concerned mainly with interaction diagrams for conversation modeling. Interaction diagrams mainly extend the OMG definition of UML sequence diagrams with the possibility to express explicitly concurrency in the sending of messages.

BPEL. Business Process Execution Language [5] is a formalism used for specifying the composition of Web services. It was created to standardize interaction logic and process automation between Web services. BPEL is a convergence of language features form WSFL and XLANG. However, this language lacks well defined semantics, which makes it difficult to reuse and compose.

OWL-S. OWL-S [6] is one of the standards for the description of Web services. It includes a process model for Web services. Each process is described by three components: inputs, preconditions and results. Results specify what outputs and effects are produced by the process under a given condition.

FSM. FSM [7] represents a powerful formalism for describing and implementing the control logic of an interaction system. They are suitable for implementing communication protocols, control interactions and describe transitional functions. FSM mainly consist of a set of transition rules. In the traditional FSM model, the environment of the machine consists of two finite and disjoint sets of signals, input signals and output signals. Also, each signal has an arbitrary range of finite possible values.

The above described representation formalisms are useful to model interactions among distributed software entities. However, not all have the same purpose and facilities. Some are good for modeling interactions formally (Petri Nets, Colored Petri Nets, Pi-Calculus and AUML), others have tools for verification and simulation (Petri Nets and Colored Petri Nets), others are good for executing and implementing composite processes (BPEL and OWL-S). But for the aim of this work we have selected FSM because they offer a simple manner of implementing transitional functions in order to compute similarity in pragmatics of represented protocols.

3 Representing Pragmatics

One of the main problems we faced when trying to evaluate the pragmatics of a software agent, was the selection of the appropriated interaction protocol representation formalism. As it was described in Section 2, there are various formalisms reported to achieve this goal. However, some are useful for modeling (UML diagrams, Pi-Calculus), some others are good for executing processes (BPEL, OWL-S), some are good for simulating interaction processes (Petri Nets and Colored Petri Nets), but we needed to use a formalism easy to implement and therefore to compute. We also needed a formalism which would help us in the automatic discovering of functionalities, which is our work in progress. Thus, we selected FSM for this reason.

A **FSM** is a tuple (S, *I*, *O*, *ft*, *fo*, *s0*), where
S is a finite set of states,
I is the set of inputs,
O is the set of outputs,
ft is the transitional function,
fo is the output function and
s0 represents the output state.

For the purpose of our work we have adapted this FSM definition in order to represent pragmatics of interaction systems. In particular we have adapted the transition functions to represent initial states, and final states generated by an incoming message.

The methodology we followed for extracting and representing pragmatics from a software entity is described next.

a) Acquisition of software entities protocols and communication messages details. Currently this process is executed manually through a Web-based environment.

b) FSM drawing. This process requires extensive human interaction, because we need to build FSM in order to obtain transition functions from them.

c) Transition functions generation. This process consists of analyzing the FSM diagrams to identify arcs and define the transitional functions.

In order to compute pragmatic similarity between different interaction protocols we established a common set of states, to allocate all the messages and transition functions in those states. Therefore, we adopted the proposal of Müller [8]. Müller specifies that any communication protocol consists of three general states: ***start***, ***react*** or ***complete*** depending on the moment in the FSM when the primitive is issued, but we added another the ***modify*** state to be more specific about a conversation between software entities.

4 An Example

In this section we present an example to show the applicability of our approach. The objective of this case is to represent and measure similarity among a set of interaction protocols from three different software entities using their communication message details.

Given a DIS integrated with three software entities e_1, e_2, and e_3, each with its own interaction protocol ***IP***.

$$DIS = \{ IPe_1, IPe_2, IPe_3 \}$$

For each IP there is a set of primitives which are used as communicative acts to send and receive messages among them. The set of interaction primitives for each IP are as follows.

IPe_1 = {*Initial_Offer, RFQ, Accept, Reject, Offer, Counter_Offer*}

IPe_2 = {*CFP, Propose, Accept, Terminate, Reject, Acknowledge, Modify, Withdraw*}

IPe_3 = {*Requests_Add, Authorize_Add, Require, Demand, Accept, Reject, Unable, Require-for, Insist_for, Demand_for*}

Based on the set of primitives described we manually generated the FSM and defined the set of transition functions to compute similarity. In Fig. 1 we present the state transition diagram of interaction protocol of software A. In Table 1 we describe the set of transition functions of software A.

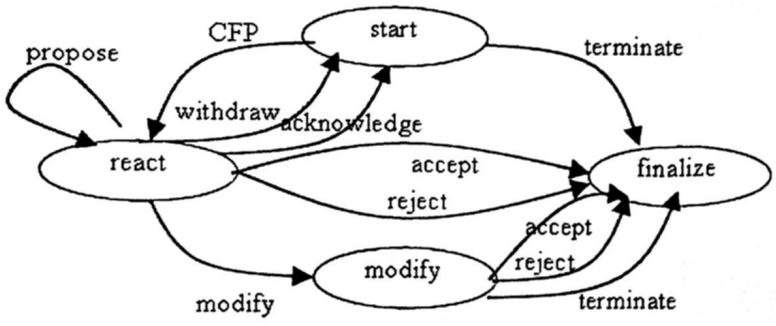

Fig. 1. Interaction protocol for software A

Table 1. Transition functions of software A

Transition functions of interaction protocol of software A
ft(start, CFP) = react
ft(react, Propose) = react
ft(react, Acknowledge) = start
ft(react, Modify) = modify
ft(react, Withdraw) = start
ft(react, Reject) = finalize
ft(react, Accept) = finalize
ft(modify, Reject) = finalize
ft(modify, Accept) = finalize
ft(react, Terminate) = finalize
ft(start, Terminate) = finalize

Fig. 2 shows the state transition diagram of software B. In Table 2 we present the transition functions of interaction protocol of software B.

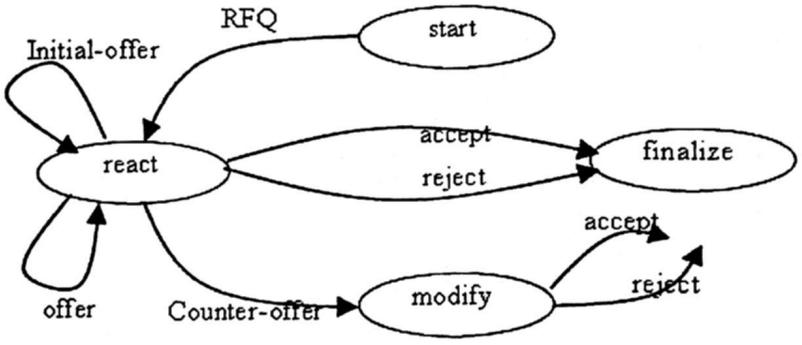

Fig. 2. Interaction protocol of software B

Table 2. Transition functions of software B

Transition functions of interaction protocol of software B
ft(start, RFQ) = react
ft(react, Initial-offer) = react
ft(react, Counter-offer) = modify
ft(react, Offer) = react
ft(modify, Accept) = finalize
ft(modify, Reject) = finalize
ft(react, Reject) = finalize
ft(react, Accept) = finalize

Fig. 3 presents the state transition diagram of software C. Table 3 shows the transition functions of interaction protocol of software C.

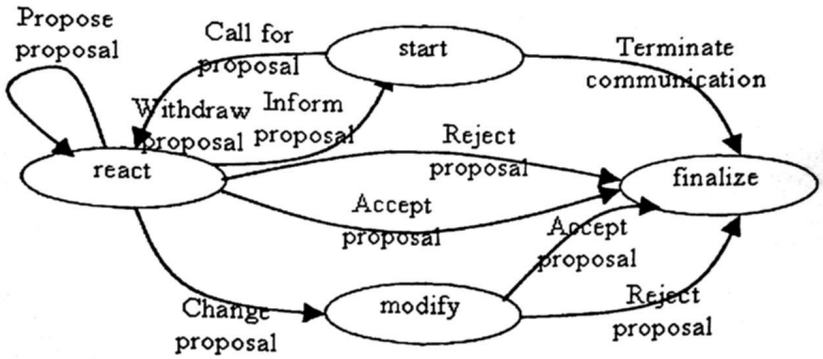

Fig. 3. Interaction protocol of software C

Table 3. Transition functions of software C

Transition functions of interaction protocol of software C
ft(start, Call for proposal) = react
ft(react, Propose proposal) = react
ft(react, Inform Proposal) = start
ft(react, Change Proposal) = modify
ft(react, Withdraw Proposal) = react
ft(react, Reject Proposal) = finalize
ft(react, Accept Proposal) = finalize
ft(modify, Reject Proposal) = finalize
ft(modify, Accept Proposal) = finalize
ft(start, Terminate Communication) = finalize

4.1 Computing Similarity

To compute pragmatic similarity we need first to calculate the total number of different interaction links and identify the set of pairs of software entities that will interact among them.

a) Number of Interaction links

Considering a set of n software entities, the possible number of peer to peer interaction links among them is n^2. However, as we are evaluating heterogeneity, we need to extract the number of interaction links where software entities are equal, which is n. We also considered that a interaction link between software entities (a, b) has the same heterogeneity as an interaction link of

software entities (b, a), thus we reduced the number of different interaction links dividing by 2.

$$CL = (n^2\text{-}n)/2$$
$$CL = (3^2\text{-}3)/2 = 3 \quad (1)$$

B) Set of Different Interaction links

Considering a set of software entities, the set of different interaction links is given by:

$$\boldsymbol{DCL} = \{ (a_1, a_2), (a_1, a_3), \ldots, (a_i, a_j) \}$$
$$\boldsymbol{DCL} = \{ (A, B), (A, C), (B, C) \} \quad (2)$$

Algorithm for Computing Pragmatic Similarity

Our algorithm is based on the equivalence definition of FSM. Given two FSM, two states that belong to each FSM are said to be equivalent if for a given initial state and a given input message, the resulting state is the same. We adapted this definition, but we are considering that the input messages may be syntactically different, and then we are computing similarity among these different messages as follows: if their initial states and their final states are equal, then the syntactically different messages are pragmatically similar. To compute pragmatic similarity we implemented an array-based algorithm. For our case study we implemented three arrays, each with three columns which represent: the initial state, the input primitive and the final state. The algorithm is executed for each different communication link (ai, aj), where ai represents the array of agent i.

The result of this algorithm is a set of relations, which will help as the basis for a translation approach solution.

```
For each transition function ft of a_i
 For each transition function ft of a_j
  If (a_i[initial-state] is equal to a_j[initial-state])
     and (a_i[final-state] is equal to
             a_j[final-state])
             if (a_j[input-primitive]
                is-different-syntactically to
                 a_j[input primitive])
                   a_i[input-primitive]
```

```
is-similar-pragmatic to
a_j[input primitive]
```

After obtaining the resulting set of similar functions, we have to evaluate them, in order to check inconsistencies. We defined only similar pragmatic relations for primitives that are syntactically different. We did not established differences as relations, because this kind of relations will not support interoperability. However, they are important to measure heterogeneity and to propose another solution based on a learning approach. Results of this process are shown in Tables 4, 5 and 6. To define relations we used the form:

$$\textbf{\textit{REL}}(S_i, P_i, S_j, P_j)$$

where

S_i is the software issuer of primitive P_i
S_j is the software issuer of primitive P_j

Table 4. Set of relations of interaction link between software entities A and B

IL(A, B)
REL(A, CFP, B, RFQ)
REL(A, Propose, B, Initial_Offer)
REL(A, Modify, B, Counter_offer)
REL(A, Propose, B, Offer)
REL(A, Terminate, B, Reject)
REL(A, Terminate, B, Accept)

Table 5. Set of relations of interaction link between software entities A and C

IL(A, C)
REL(A, CFP, C, Call for proposals)
REL(A, Propose, C, Propose proposal)
REL(A, Modify, C, Change proposal)
REL(A, Withdraw, C, Withdraw proposal)
REL(A, Acknowledge, C, Inform proposal)
REL(A, Accept, C, Accept proposal)
REL(A, Reject, C, Reject proposal)
REL(A, Terminate, C, Terminate communication)

Table 6. Set of relations of interaction link between software entities B and C

IL(B, C)
REL(B, RFQ, C, Call for proposals)
REL(B, Offer, C, Propose proposal)
REL(B, Counter_offer, C, Change proposal)
REL(B, Accept, C, Accept proposal)
REL(B, Reject, C, Reject proposal)
REL(B, Initial_Offer, C, Propose proposal)

4.2 Pragmatic Similarity Measure

Another important result is a pragmatic similarity measure, which will help to analyze more precisely the level of similarity of a set of software entities. In this section we describe this measure.

The pragmatic similarity measure is a ratio which results from dividing the number of equivalent functions by the total number of transition functions from participating software entities protocols.

a) Number of transition functions

The total number of transition functions is obtained from the sum operation of all sets of transition functions.

$$NTF = |TFa_1 + TFa_2 + ... + TFa_n| \quad (3)$$

b) Number of equivalent functions

The total number of equivalent functions (NEF) results from the sum of the resulting set of the algorithm.

c) Pragmatic similarity

The pragmatic similarity results from dividing the number of equivalent functions by NFT, which is the ratio that will serve as an indicator for evaluating pragmatic similarity.

$$Pragmatic\ similarity = NEF / NFT \quad (4)$$

Table 7. Pragmatic similarity measure between each interaction link

IL	*NEF*	*NTF*	*Pragmatic Similarity*
A, B	6	19	.31
A, C	8	21	.38
B, C	6	18	.33

The resulting pragmatic similarity ratios for the three interaction links is shown in Table 7.

Therefore a ratio of 1 indicates a fully pragmatic similarity between software entities, while a ratio of 0 indicates impossibility of interoperation between both software entities.

5 Interaction Environment

To evaluate our approach we implemented an interaction environment populated with the set of software entities described in Section 4. We also implemented a translator module which is invoked whenever is necessary. The general architecture for the execution of interaction protocols is illustrated in Figure 4. This architecture has an intermediary program which is responsible for initialization of interaction processes, sending and receiving messages form both software entities, and recording the communication until an ending message is issued by any of the participants.

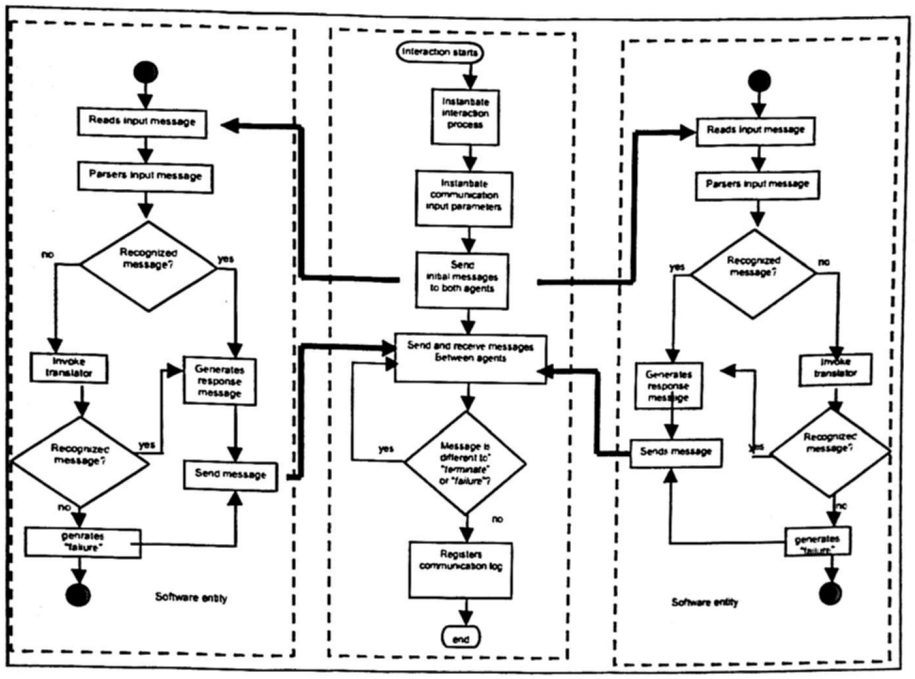

Fig. 4. Interaction Web-based architecture

The software entities are programmed by their respective owners to define their preferences and interaction strategies. For example, a seller software entity will be programmed to maximize his profit, establishing the lowest acceptable price and the

desired price for selling. In contrast, a buyer software entity is seeking to minimize his payment.

The translator module is invoked whenever the software does not recognize a message from the other software. The translator module was implemented using Jena[1], a framework for building Semantic Web applications. It provides a programmatic environment for OWL, including a rule-based inference engine. For the description and execution of communication processes, we used BPEL4WS. BPEL4WS defines a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners. BPEL4WS represents a convergence of the ideas in the XLANG and WSFL specifications. Both XLANG and WSFL are superseded by the BPEL4WS specification. The interaction with each partner occurs through Web service interfaces, and the structure of the relationship at the interface level is encapsulated in what we call a partner link. The BPEL4WS process defines how multiple service interactions with these partners are coordinated to achieve a business goal, as well as the state and the logic necessary for this coordination.

5.2 Translator Functionality

The translator acts as an interpreter of different software entities. The translator module first reads the input parameters, and then opens a connection to the Ontology to make queries. When the connection has been established it narrows the search by selecting only instances of the same *Type* of the incoming "message". It then searches for and retrieves all primitives that belong to the "receiver" software entity in the same *Type*. And finally it makes a comparison to find a similar primitive, when this process ends it returns the equivalent primitive; in other case it returns a failure message.

A misunderstanding event occurs when a software entity receiving a message, compares it to its own message knowledge base, and acknowledges that the received message is not in his base, and then invokes the translator to find the relation with its own messages.

6 Experimentation Results

We executed a series of tests with these software entities. In this section we present one experiment of a set of 30 communication executions between software entities A and B. The result of this experiment is shown in Table 8.

Table 8 shows that for the first set of executions many ended because of misunderstandings. For the second set of executions we can appreciate a reduction in the number of misunderstandings, although the problem remains, some communications are still ending due to this problem. For this case we suggest using a learning approach, in order to fully eliminate the problem. However, the result is good enough to evaluate the main contribution of this paper. The set of discovered

[1] http://jena.sourceforge.net

pragmatic relations were well defined by our semi-automatic approach. The Ontology was populated with these primitives and relations among them, and when integrated to the general environment it solved the main problem of our work: interoperability.

Table 8. Experimental results

	Result	Result
Test number	**No translation**	**Using translation**
1	notUnderstood	Accept
2	notUnderstood	Reject
3	notUnderstood	Accept
4	notUnderstood	Accept
5	notUnderstood	Accept
6	notUnderstood	Accept
7	notUnderstood	Reject
8	notUnderstood	Reject
9	notUnderstood	Accept
10	notUnderstood	notUnderstood
11	notUnderstood	Accept
12	notUnderstood	Accept
13	notUnderstood	Accept
14	notUnderstood	Accept
15	notUnderstood	Accept
16	notUnderstood	Accept
17	notUnderstood	Reject
18	notUnderstood	Accept
19	notUnderstood	Accept
20	notUnderstood	Accept
21	notUnderstood	Reject
22	notUnderstood	Accept
23	notUnderstood	Accept
24	notUnderstood	notUnderstood
25	notUnderstood	Reject
26	notUnderstood	Reject
27	notUnderstood	Reject
28	notUnderstood	Accept
29	notUnderstood	Reject
30	notUnderstood	notUnderstood

Figure 5 shows the graphical results of this experiment. The first set of bars represents results of communications without translations, and the second set of bars represent results of communications invoking the translator.

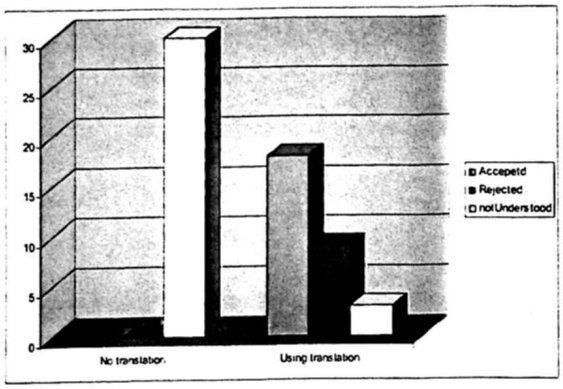

Fig. 5. Experimental results

7 Related Work

There is currently a research effort dealing with the problem of modeling Web service processes; in particular, some of them have proposed conformance and interoperability measures between two composed Web service processes. In [8] authors present the behavior equivalence concept, which states that two executable processes are equivalent if they can be transformed by a set of transformation rules.

Baldoni et al. [9], present a calculus to allow the automatic construction and update of compatible services. They define formally the compatibility, interoperability, conformance and substitutability between Web services concepts. They present a set of derivation rules to verify the conformance level and to construct new conformant and interoperable services.

Van der Aalst et al. [10] present the concept of process equivalence between two process models based on their observed behavior. They provide a measure of equivalence with a Lumber ranking from 0 to 1. This represents a more precise measure.

In this work we are dealing with agent communication protocols, which have similar functionality compared to composite Web service processes. However, communication protocols follow specific interaction rules; for example, an agent following a conversation protocol never issues a completer communicative act before receiving a starter communication. While Web service processes may follow different flows depending on the problem or objective they are following. Therefore, even

though we consider important these works, they are not directly transferable to our domain.

8 Conclusions

In this paper we have presented how to compute pragmatic similarity in DIS. Our approach is novel, because it presents an algorithm based on the analysis of FSM transition functions, which in turn are capable of being implemented and compared to obtain pragmatic relations and pragmatic similarity measures. In contrast to the semantic approach, which has proven good results for data, vocabularies and information sources, we propose a semi-automatic approach for comparing software pragmatics. The result of this analysis helps the developer to measure pragmatic similarity and identify pragmatic relations. The set of defined relations were implemented in an ontology-based translator which showed better results in communication environments when the translator was invoked.

The pragmatic similarity measure will help developers to decide and propose a good solution, for example, translation, learning or reprogramming all software entities to be fully interoperable among them.

As the Semantic Web has been evolving we are facing new requirements, such as discovering semantic and pragmatic relations from heterogeneous sources. This is a promising research area, because nowadays there is a tremendous amount of legacy software which in turn will require to be incorporated transparently giving the idea of an integral solution independently of the inherent heterogeneity inside their logics or protocols.

FSM is a good formalism for representing communication scenarios between software entities, in particular they are suitable to compare interaction protocols and identify the state of a conversation in a DIS environment.

Finally, we are sure that our pragmatic computing approach can be applied in other application areas such as Web service discovery, process engineering, and program comparison or application integration, which have in common that they provide functionality information similar to protocols in communication scenarios.

References

1. Petri, Carl A. Kommunikation mit Automaten. Ph. D. Thesis. University of Bonn, 1962.
2. R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou, and You Peng. Modeling agent conversations with colored petri nets. In working notes of the Workshop on Specifying and Implementing Conversation Policies, pp. 59-66, Seattle, Washington, 1999.
3. R. Milner. Communicating and Mobile Systems: The Pi-Calculus, Cambridge Univ. Press, 1999.
4. Odell J. Van Dyke Parunak, H., and Buer, B. Representing agent interaction protocol UML. In the first international workshop, AOSE 2000 on Agent Oriented Soft Engineering, pp. 121-140, Springer Verlag, Berlin, 2001.BPEL, Business process Execution Language for Web Services. Version 1.1, May, 2003.

5. DAML-S: Web service description for the semantic Web. In Proceedings of the first International Web Conference, ISWC, 2002.
6. J.E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to Automata Theory, Languages and Computation, Addison Wesley, 2001.
7. Müller, H. J., 1996. Negotiation Principles, Foundations of Distributed Artificial Intelligence. *In G.M.P. O'Hare, and N.R. Jennings*, New York: John Wiley & Sons.
8. Köning D., Lohmann. N., Moser, S., Stahl, C., Wolf, K. Extending the Compatibility Notion for Abstract WS-BPEL Processes. En las memorias de la 17 Conferencia Internacional de la World Wide Web, (WWW 2008), Beijing, China, 2008.
9. Baldoni, M., Baroglio, C.,Patti, V., Chopra, A., Desai, N., Munindar, S. Calculi for Service Conformance and Interoperability. Draft paper, 2008.
10. Van der Aalst, W.M.P., de Medeiros, A.K. Alves, Weijters, A.J.M.M., Process Equivalence: Comparing Two Process Models Based on Observed Bahvior. LNCS, Business Process Management, Vol. 4102, 2006.