A Language to Specify the Interaction Considering Agents, Environment, and Organization

Maicon R. Zatelli, Jomi F. Hübner
Department of Automation and Systems Engineering
Federal University of Santa Catarina (UFSC)
Florianópolis, SC, Brazil
{*maicon,jomi*}*@das.ufsc.br*

***Abstract*—Interaction is a subject widely investigated in multi-agent systems (MASs), but there are still some open issues. Beyond the interaction usual between agents, we can conceive other kinds, like the interaction between agents and environment, or between agents and organization. These other kinds allow us to consider several situations that are not limited to speech acts. For example, the interaction between agents and environment allow us to define actions and events in interaction protocols, which would not be possible to represent with just the concept of speech act. In this paper we propose a language to specify the interaction considering the environment, organization, and agents. We also present a sketch of a dynamic of execution and some examples of protocols.**

***Keywords*-interaction; language; AEIO; environment; organization**

I. Introduction

This work is based on the AEIO approach (*Agent*, *Environment*, *Interaction*, *Organization*) [1], which conceives a multi-agent system (MAS) as composed of four basic components: agents, environment, interaction, and organization. Therefore, an MAS is not only based on the existence of agents, but there are other elements equally important. The developer should be able to see each of these parts clearly and separately.

Nowadays, it already exists many works about agents, organization, and environment. There are tools to specify, develop, and execute each one. For example, an MAS developer is able to build the environment by means of CArtAgO [2], the organization by means of AGR [3], ISLANDER [4], Moise [5], and so forth, and finally, the agents by means of GOAL [6], JADE [7], Jason [8], and so on. There are also tools to link these components to work together, such as JaCaMo [9]. However, none of the current tools provide features to specify and execute the interaction considering the existence of the three other components, that is, to define the interaction between agents, between agents and environment, and between agents and organization. As a consequence, the interaction is specified inside of the other MAS components, which results in difficulties to maintain, to reuse code, to debug, to work with open systems, etc.

In [10] we presented some advantages to separate the interaction of the other MAS components. With a separated interaction component it is possible to provide tools to improve debugging, because we can monitor the MAS execution from the interaction viewpoint. Moreover, as our proposal considers the interaction with the environment by means of actions or events, it is possible to represent how the agents have to proceed to interact with the several elements in the environment. We can also improve the use of open systems, where the agents can be heterogeneous. Open systems can be improved because the agents do not need to know in advance how to interact with the several elements in the MAS, but they just need to know how to handle the interaction component. Afterwards, the agents can follow the interaction specification to interact with the other MAS elements. Therefore, the migration of the agents to other MAS is also facilitated. In addition, the proposed model helps the agents to accomplish their organizational goals. The agents usually receive the goals related to their roles, but they do not receive what they must do to achieve them. In this case, the interaction helps the agents with a well-defined sequence of steps, including actions, messages, and events, which institutionalizes how the agents should interact to achieve the goals. Finally, a separated interaction component improve the reuse of code and the maintenance in an MAS because the whole interaction code is written separately from the rest of the system, which facilitates to visualize, to locate, and to change/update the interaction code.

Since the current languages are not suited to specify the interaction with the other MAS components, in this paper, our aim is to propose a programming language to specify interaction protocols considering the organization, the environment, and the agents (section III). This language follows the interaction model introduced in [10], where an *unified* and *coherent* interaction model is proposed (section II). In the proposed language, we have added some features to represent interaction protocols for several different scenarios. Furthermore, we also present a sketch of a dynamic of exe-

The authors are grateful for the support given by CNPq, grants 140261/2013-3 and 306301/2012-1

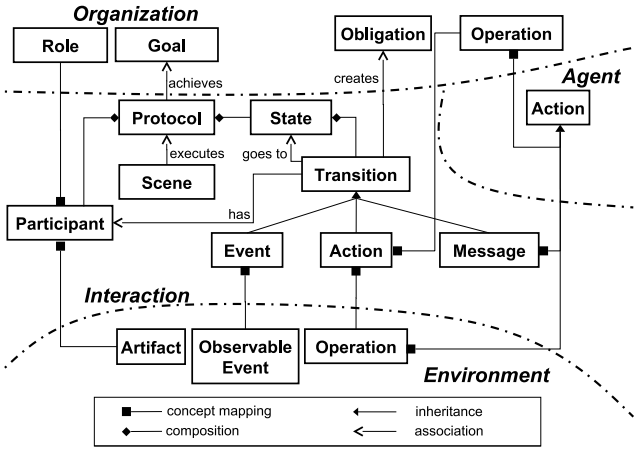

Figure 1: Conceptual model.

```
protocol        ::= "protocol" <ID> "{" description
                                        goals
                                        participants
                                        states
                                        transitions "}"
description     ::= ("description" ":" <STRING> ";")?
goals           ::= "goals" ":" (goal)+
goal            ::= <STRING> ";"
participants    ::= "participants" ":" (participant)+
participant     ::= participantId partDescription ";"
partDescription ::= ("agent" role | "artifact" type) partCardinality
partCardinality ::= ("all" | ("min" <INTEGER>)? ("max" (<INTEGER> | "+"))?)
participantId   ::= <ID>
type            ::= <STRING>
role            ::= <STRING>
states          ::= "states" ":" (state)+
state           ::= stateId ("initial" | "final")? ";"
stateId         ::= <ID>
transitions     ::= "transitions" ":" (transition)+
transition      ::= stateId "-" stateId "#" (occurrence | timeout | import)
timeout         ::= "timeout" <INTEGER> ";"
import          ::= "import" <STRING> mapping ";"
mapping         ::= "mapping" "{" (mapFromTo)+ "}"
mapFromTo       ::= participantId participantId ";"
occurrence      ::= pCardOccur "--" duty "->" pCardOccur ((trigger)+ | ";")
pCardOccur      ::= participantId ("[" <INTEGER> "]")?
duty            ::= dutyType <STRING>
dutyType        ::= ("event" | "action" | "message" "[" <ID> "]")
trigger         ::= ("trigger" pattern (":" content)? | ":" content) ";"
pattern         ::= <STRING>
content         ::= <STRING>
```

Figure 2: Language grammar.

cution (section IV) and some examples of protocols. Finally, before conclusion, we show the results and discussion with some related work.

II. Conceptual Model

This section briefly presents how the several MAS components are conceptually integrated with the interaction. Only the core ideas of the model are described here and examples will be presented in the next sections. More details can be found in [10].

In this model, the concepts of the other components were mapped onto the interaction concepts. Figure 1 shows the four MAS components and the links between the interaction and the others. In order to keep the figure clear and clean, we only show the concepts that were used in the model. The most important concept in our model is the interaction protocol, which is composed of a set of participants, transitions, states, and goals. Each transition links two states and it can be fired by an event, a message, or an action. The following paragraphs briefly explain how the other components are connected.

Protocols are linked to organizational concepts[1] in four points (top of Figure 1). When a protocol finishes successfully, an organizational goal is considered achieved. Other organizational concepts used in the interaction component are the roles, which constrain the participation of agents in the protocol; the obligations, which agents have to follow in order to accomplish the protocol; and the operations, which are the actions that some agent can perform in the organization such as adopt or leave some role.

The environment[2] also has some important concepts to be considered in the interactions. We mapped the concept of artifact onto a participant in the interaction component, which constrains the participation of artifacts in the protocol; the operations, which represent the actions that the agents can perform in the environment; and finally, the observable events, which agents can perceive in the environment such as an alarm, the color of something, etc.

In the end, the agent component provides the concepts of action, which can be some action performed in the environment or in the organization, and the message exchange, which represents the use of communicative acts to interact with the other agents.

III. A Language to Specify Interaction Protocols

In this section, we map the concepts presented in Figure 1 onto a programming language to specify interaction protocols[3]. Figure 2 presents the language grammar with its non-terminal symbols. A protocol is composed of a name, a description (represented by the non-terminal `description`), goals that will be achieved (represented by the non-terminal `goals`), participants (represented by the non-terminal `participants`), states (represented by the non-terminal `states`), and transitions (represented by the non-terminal `transitions`). We explain the main language features in detail by means of some examples.

Protocol 1 presents a first example, where the aim is to perform an election between the agents. The participation of the agents is defined in line 5, which state that they must play the role `elector` in the organization. The protocol includes the participation of a ballot box artifact to help the agents to vote in an anonymous approach (line 6).

The protocol is composed of three states (line 7): `n1`, `n2` e `n3`, where `n1` is the initial state and `n3` is the final state.

[1]Organizational concepts are explained in more details in [5], [11].

[2]Environment concepts are explained in the A&A meta-model introduced in [12].

[3]Due the lack of space we will only present the most important parts of the language.

Protocol 1 Election protocol.

```
1. protocol election {
2.   description: "Do an election";
3.   goals: "electLeader";
4.   participants:
5.     playerElector agent "elector" all;
6.     artBallotBox artifact "artifacts.BallotBox";
7.   states: n1 initial; n2; n3 final;
8.   transitions:
9.     n1 - n2 # playerElector -- action "vote(X)" -> artBallotBox
                 : ".string(X) & .is_agent(X)";
10.    n1 - n2 # timeout 30000;
11.    n2 - n3 # artBallotBox -- event "winner(Y)" -> playerElector;
12. }
```

Protocol 2 Attending protocol.

```
1. protocol attending {
2.   description: "Serve a costumer";
3.   goals: "chooseSeller";
4.   participants:
5.     playerCustomer agent "client";
6.     playerSeller agent "seller" all;
7.   states: k1 initial; k2; k3 final;
8.   transitions:
9.     k1 - k2 # playerCustomer -- message[tell] "needSeller" -> playerSeller;
10.    k2 - k3 # import "election.ptl" mapping { playerSeller playerElector; };
11. }
```

Protocol 3 Composition between the attending and election protocols.

```
1. protocol attending {
2.   description: "Serve a customer";
3.   goals: "chooseSeller";
4.   participants:
5.     playerCustomer agent "client";
6.     playerSeller agent "seller" all;
7.     artBallotBox artifact "artifacts.BallotBox";
8.   states: k1 initial; k2; n2[k2]; k3 final;
9.   transitions:
10.    k1 - k2 # playerCustomer -- message[tell] "needSeller" -> playerSeller;
11.    k2 - n2[k2] # playerSeller -- action "vote(X)" -> artBallotBox
                     : ".string(X) & .is_agent(X)";
12.    k2 - n2[k2] # timeout 30000;
13.    n2[k2] - k3 # artBallotBox -- event "winner(Y)" -> playerSeller;
14. }
```

The available transitions from state `n1` are those defined in lines 9 and 10. The first one can be triggered only by agents participating as `playerElector` in the protocol by doing the action `vote(X)` on the artifact `artBallotBox` (the ballot box). Moreover, when the protocol is in the state `n1` an obligation to perform the action `vote(X)` is created for the agents playing `elector`. Although created from a fact in the interaction component, this obligation exists in the organizational component of the MAS.

Note that a transition between `n1` and `n2` is defined with a `timeout` (line 10). The `timeout` is important in situations where the temporal constraints are fundamental, such as the time that an agent must wait for the proposals of the others in an auction. Moreover, the liveness in the protocol can be improved by means of a timeout, that is, the protocol will always achieve a final state.

The last transition (line 11) of the protocol defines that the participant `artBallotBox` must count the votes and emits an observable signal named `winner(Y)`, where `Y` is the winner name. With the successful termination of the protocol, the goal `electLeader` is achieved in the organization (line 3).

A second example of protocol (Protocol 2) describes the situation where a virtual agent decides to buy something in a website. The new protocol has three states (line 7): `k1`, `k2`, and `k3`, where `k1` is the initial state and `k3` is the final state. The first transition (line 9) defines that the agent that is playing the participant `playerCustomer` must send a message to the agents that are playing the participant `playerSeller` telling them that it needs some seller. The next transition (line 10) defines that the sellers must perform an election to decided which one will attend to the client. This transition has an `import` directive, which allows the composition of protocols. The address of the sub-protocol and a mapping between the participants of both protocols are necessary and since the election protocol was already specified before, then the composition allows reusing it.

The transition with the import directive (line 10) notifies the interpreter to establish a link between the state `k2` and the initial state of the election protocol. All final states of the election protocol are mapped to the state `k3`. The other states of the election protocol are just renamed to avoid the clash of identifiers. For example, the state `n2` of the election protocol is renamed to `[k2]` because the state `n2` is the result of the composition between the attending protocol and the election protocol by means of the state `k2`. The state names should be chosen carefully to keep the protocol understandable.

Another element that exists in the `import` directive is a `mapping` between the participants that exist in the attending protocol onto the participants that exist in the election protocol. In the example, the `playerElector` in the election protocol will be replaced by the participant `playerSeller` of the attending protocol while the participant `artBallotBox` is preserved. The description and goals of the election protocol are discarded due to the composition. The result of the composition between both protocols is presented in Protocol 3.

The language also provides two different kinds of cardinality: the participant cardinality and the transition cardinality. The former is related to the number of necessary entities to play some participant in the protocol. The latter is related to the number of entities that are necessary to perform the duty specified in some transition.

The participant cardinality is represented by means of the non-terminal symbol `partCardinality`. Broadly speaking, it defines the minimum and the maximum number of entities that must play some participant. While the minimum cardinality is represented by a number, the maximum cardinality also can be a number or even the `all` or + directives. The `all` directive informs the interpreter that all agents that are playing some organizational role or all artifacts that are of some type must play the participant. The + directive informs the interpreter that the number of required entities must be between one and the total of entities that the `all` directive calculates.

In contrast to the participant cardinality, the transition

cardinality allows just numeric values. Its aim is to define a minimum number of entities that must perform or undergo some occurrence specified in the transition. The non-terminal `pCardOccur`, next to the transition participant (between square brackets), represents the transition cardinality. By default, the omission of the transition cardinality results in the use of the same value that exists in the participant cardinality, meaning the participant cardinality and the transition cardinality are the same. There are four situations that can be represented by means of the transition cardinality. The first situation (`player1[1] -- action "foo" -> player2`) informs that at least *one* agent (left side) needs to execute some action in *all* artifacts (right side), that is, *all* artifacts must undergo at least *one* action. The second one (`player1 -- action "foo" -> player2[1]`) informs that *all* agents must execute an action in at least *one* artifact. The third one (`player1[1] -- action "foo" -> player2[1]`) informs that at least *one* agent is necessary to execute an action in at least *one* artifact. The latter (`player1 -- action "foo" -> player2`) informs that *all* agents must execute an action in at least *one* artifact and *all* artifacts must undergo at least *one* action.

Finally, the non-terminal `duty` defines what must be performed to fire the transitions and each transition may have several different verifications (represented by the non-terminal `trigger`) to make sure whether the occurrence is valid to fire it. The non-terminal `trigger` is composed of an expression to evaluate the occurrence pattern (represented by the non-terminal `pattern`) and an expression to evaluate the occurrence content (represented by the non-terminal `content`). If the `pattern` is omitted, the expression defined in the non-terminal `duty` will be considered as the `pattern`. For example, consider the action `vote(X)` presented in Protocol 1. The agent receives this obligation and it has to perform the action `vote`. As the pattern is omitted, the expression specified in the `duty` (i.e. `vote(X)`) is used as the pattern. Next to the symbol `:` (line 9), it is defined the expression to evaluate the content of the action. Suppose the agent tries to execute something like `vote("Ana",22)`. This action is not valid because it does not unify with the pattern `vote(X)`, then the action is discarded. However, suppose that the agent performs the action `vote(22)`. This action follows the pattern because it unifies the pattern (with `X = 22`), however the action is invalid because 22 is not a String as required by the `content`. Finally, suppose the agent tries to execute the action `vote("Ana")`. In this case, we have `X = "Ana"` and `"Ana"` is a `String`. In the case where Ana is also an agent, the action is valid to fire the transition.

IV. Sketch of the Dynamic of Execution

In this section, we briefly present the dynamic of execution. In an MAS with organization, the agents usually adopt a role and start working to accomplish the organizational goals related to its role. In the case where the goal has a protocol specified, the agent can instantiate it in order to achieve the goal. After the instantiation, the agent must ask the other agents to join the scene and must add the artifacts that will attend the scene. For example, in a scene of the Protocol 1, it is necessary the definition of the agents that will play the participant `playerElector` and also the artifact the will play the participant `artBallotBox`. Once the participating artifacts and agents are defined, the agent can start the execution of the scene. When the scene starts, it enables the transitions of the initial state and also creates the related obligations into the organization. Afterwards, the agent can follow the obligations in order to accomplish the protocol. For example, an obligation is created into the organization for all electors that are attending the scene of the Protocol 1 to obligate them to perform their votes when the scene starts.

Throughout the MAS execution, several occurrences (messages, actions, events) are intercepted and sent to the scenes. Then, the occurrences are processed in order to check whether they are valid to fire some enabled transition. For example, the agent Bob can execute the action `vote("Ana")` and this action must be intercepted and added in a queue to be processed afterwards.

The evaluation process occurs as following. While the invalid occurrences must be discarded, the valid ones must be added in a set and when the occurrences satisfy the cardinality of some enabled transition, the transition can fire and make the scene achieve a new state. The cardinality of a transition is satisfied when two conditions are true: (i) the number of valid occurrences that have the source entities that are playing the responsible participant of the transition is greater or equal to the cardinality specified for the responsible participant of the transition; (ii) the number of valid occurrences that have the target entities that are playing the target participant of the transition is greater or equal to the cardinality specified for the target participant of the transition.

For example, in a scene that executes the election protocol (Protocol 1), all electors are obligated to perform their votes. Thus, the agents must execute the action `vote` on the ballot box artifact. Each `vote` action that was performed is processed according to their informations, such as the agent that performed the vote, the description of the action vote with its parameters, and the ballot box that was used. Suppose the vote actions of all electors were processed and they were considered valid actions, however the vote action performed by the agent Bob still needs to be processed. Considering the same action `vote("Ana")` used in section III, note that it is valid to fire the transition between the states `n1` and `n2`, because it respects the validation expression defined in the transition. Also, suppose it has the responsible participant as an elector in the scene, the target participant is a ballot box in

the scene, and this action is not repeated. As this is the only action that was lacking to complete the cardinality defined in the transition (*all* electors must execute an vote action in *one* ballot box), considering this action, the transition `n1-n2` can fire and make the scene achieve the next state (`n2`).

A different situation can happen when some occurrence is valid but no transition has the cardinality satisfied to fire. For example, this situation could happen in some election with more than one elector. Even if some agent executes the action `vote("Ana")`, which is a valid vote action, it will be necessary to wait for all vote actions from the other electors to fire the transition between the states `n1` and `n2`.

Another way to achieve a new state is when some timeout happens. If no transition is fired in time, then a timeout can happen and the next state is pointed by the timeout transition. For example, in the election protocol (Protocol 1), if not all electors vote in 30 seconds, defined by the second transition between the states `n1` and `n2`, then a timeout happens and the scene achieves the state `n2`, pointed by the timeout transition.

Finally, when some transition is fired, the protocol can achieve a final state and then the organizational goals related to the protocol must be satisfied. For example, when some scene of the election protocol (Protocol 1) achieves the state `n3`, which is stated as final state, the goal `electLeader` must be satisfied in the organization.

V. Results and Discussion

In order to validate our approach we have integrated it into JaCaMo platform [9]. JaCaMo is a project that aims to permit the developer to consider each one of the MAS components as first class abstractions. Although the agent, environment, and organization components are already considered by this platform, the interaction component was not properly integrated.

Our main contribution is a programming language to specify interaction protocols considering the agents, environment, and organization. The aim is to institutionalize how the agents must interact with the different elements in an MAS to achieve the organizational goals by means of protocols. When an agent adopts some role in the organization, the agent receives the list of goals that it needs to achieve. In order to achieve them, the agent may look at the interaction component for a protocol that achieve them. It is useful since, sometimes, the agents may not know how to proceed to achieve their goals, then the protocol helps the agents in this situation by means of a well-defined sequence of steps.

The integration with the organization also helps the agents to search for partners to cooperate. They can do it reading the roles in the organization and the agents that are playing each role. Other important organization concept are the obligations that are useful to help the agents to follow the protocols. The obligations facilitate the agent programming and allow the agents to reason about them, specially whether the agents already can handle with organizational obligations. Moreover, such organizational mechanisms allow us to create punishment and reward mechanisms to prevent malicious behavior and reward the agents with good performances.

However, even in simple and closed systems, where the organization may not be necessary, our model needs an organization, which could require more time to develop the MAS. Indeed, our proposal is focused in more complex MAS, composed of agents, environment, and organization. Our aim is to integrate these components by means of the interaction and explore the advantages of this kind of MAS.

Other differential of our proposal is to regard the interaction with the environment, which allows the representation of more scenarios. For example, the language allows the specification about how the agents have to proceed to interact with the artifacts by means of actions and events.

The language also provides features like composition, cardinality, timeout, which allow the representation of several scenarios. These features allow improvements to reuse code, because it is possible to build more complex protocols by combining simple protocols; to represent real-time situations, because of the timeout mechanism; to comprehend the protocol, since the language is suited to conceive interaction protocols and it is written as a whole; etc.

The model can also be used in open systems where the agents can be heterogeneous. The separation of the interaction component of the other MAS parts endows the system with this capacity. The agents can join the MAS, play some role and read the interaction protocols in runtime. In order to do it, the agent must know how to follow protocols by means of obligations and then, the agent is able to learn new protocols and to interact with other agents or even with the environment. In addition, we can change the protocol specification without change the agent code. Indeed, it would be necessary for an agent to communicate with other agents if it does not understand what is the meaning of some protocol step that was modified. Finally, even in the case of open and heterogeneous MAS, a global behavior can be defined for the overall system.

A. Related Work

As already presented in [10], the interaction in MAS has several different approaches. Most of approaches do not consider the interaction between the agents and the environment or between the agents and the organization [3], [4], [11], [13]–[18], but some of them already try to conceive an interaction model that handles the interaction with the other components [19]–[23]. However, their aim is different than ours. For example, their interaction specification is conceived to be handled by humans during the MAS design and do not allow the agents to read it (or eventually to change it) at run-time. Furthermore, in [21], [22], the organization is considered in a simplified version, just with

roles, because their focus is to deploy a different way to use the environment during the interactions.

The MERCURIO framework [23], a very similar work to ours, focuses on interaction regarding agents and environment. The environment conception considers the actions performed by the agents and the event that the agents may sense. The limitation of MERCURIO is related to the organizational component. The roles in the interaction are not strongly connected with the roles existing in the organization. The existence of the other organizational concepts is not considered either. Indeed, the aim of MERCURIO is to deploy the interaction with the environment.

On the other hand, MAS-ML [19] and O-MaSE [20] are a modeling language and a methodology, respectively, that consider the interaction integration with the three other components. Both approaches are conceived for the specification phase, not regarding the implementation and execution phases. For example, these methodologies do not provide a feature to generate the interaction code. Therefore, our work contributes to fill the gap between this specification and the implementation.

REFERENCES

[1] Y. Demazeau, "From interactions to collective behaviour in agent-based systems," in *Proc. of EuroCogSci, Saint-Malo*, 1995, pp. 117–132.

[2] A. Ricci, M. Viroli, and A. Omicini, "CArtAgO: An infrastructure for engineering computational environments in MAS," in *Proc. of E4MAS*, D. Weyns, H. V. D. Parunak, and F. Michel, Eds., AAMAS 2006, Hakodate, Japan, 2006, pp. 102–119.

[3] J. Ferber, O. Gutknecht, and F. Michel, "From agents to organizations: An organizational view of multi-agent systems," in *Proc. of AOSE*. Springer, 2003, pp. 214–230.

[4] M. Esteva, B. Rosell, J. A. Rodriguez-Aguilar, and J. L. Arcos, "Ameli: An agent-based middleware for electronic institutions," in *Proc. of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, ser. Proc. of AAMAS. Washington, DC, USA: IEEE Computer Society, 2004, pp. 236–243.

[5] J. F. Hübner, J. S. Sichman, and O. Boissier, "A model for the structural, functional, and deontic specification of organizations in multiagent systems," in *Proc. of SBIA*. London, UK: Springer, 2002, pp. 118–128.

[6] K. V. Hindriks, "Programming rational agents in GOAL," *Multi-Agent Programming: Languages and Tools and Applications*, pp. 119–157, 2009.

[7] L. Braubach, E. Pokahr, and W. Lamersdorf, "Jadex: A bdi agent system combining middleware and reasoning," in *Ch. of Software Agent-Based Applications, Platforms and Development Kits*. Birkhaeuser, 2005, pp. 143–168.

[8] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*. Liverpool: Wiley, 2007.

[9] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi, "Multi-agent oriented programming with JaCaMo," *Science of Computer Programming*, 2011.

[10] M. R. Zatelli and J. F. Hübner, "A unified interaction model with agent, organization, and environment," in *Anais do IX Encontro Nacional de Inteligência Artificial (ENIA@BRACIS)*, Curitiba, Brazil, 2012.

[11] V. Dignum, J. Vázquez-salceda, and F. Dignum, "Omni: Introducing social structure, norms and ontologies into agent organizations," in *Proc. of PROMAS*. Springer, 2004, pp. 181–198.

[12] A. Omicini, A. Ricci, and M. Viroli, "Artifacts in the A&A meta-model for multi-agent systems," *Autonomous Agents and Multi-Agent Systems*, vol. 17, pp. 432–456, 2008.

[13] E. Platon, N. Sabouret, and S. Honiden, "Overhearing and direct interactions: point of view of an active environment, a preliminary study," in *Proc. of E4MAS*. Springer, 2005, pp. 121–138.

[14] D. Keil and D. Q. Goldin, "Indirect interaction in environments for multi-agent systems," in *Proc. of E4MAS*, 2005, pp. 68–87.

[15] J. Saunier and F. Balbo, "Regulated multi-party communications and context awareness through the environment," *Multiagent Grid Syst.*, pp. 75–91, 2009.

[16] O. Boissier, F. Balbo, and F. Badeig, "Controlling multi-party interaction within normative multi-agent organizations," in *Proc. of MALLOW*, 2010, pp. 17–32.

[17] A. Hübner, G. P. Dimuro, A. C. R. Costa, and V. L. D. Mattos, "A dialogic dimension for the Moise+ organization model," in *Proc. of MALLOW*, 2010, pp. 21–26.

[18] M. P. Singh, "Information-driven interaction-oriented programming: BSPL, the blindingly simple protocol language," in *Proc. of AAMAS*, 2011, pp. 491–598.

[19] V. T. Silva, R. Choren, and C. J. P. de Lucena, "A uml based approach for modeling and implementing multi-agent systems," in *Proc. of AAMAS*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 914–921.

[20] S. A. DeLoach and J. L. Valenzuela, "An agent-environment interaction model," in *Proc. of AOSE*. Berlin, Heidelberg: Springer, 2006, pp. 1–18.

[21] E. Oliva, M. Viroli, A. Omicini, and P. Mcburney, "Argumentation and artifact for dialogue support," in *Argumentation in Multi-Agent Systems*, ser. LNAI. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 107–121.

[22] Y. Kubera, P. Mathieu, and S. Picault, "Interaction-oriented agent simulations: From theory to implementation," in *Proc. of ECAI*. Patras, Greece: IOS Press, 2008, pp. 383–387.

[23] M. Baldoni, C. Baroglio, F. Bergenti, E. Marengo, V. Mascardi, V. Patti, A. Ricci, and A. Santi, "An interaction-oriented agent framework for open environments," in *Proc. of AI*IA*. Berlin, Heidelberg: Springer, 2011, pp. 68–79.